Ingeniaritza Konputazionala eta
Sistema Adimentsuak Unibertsitate Masterra

Máster Universitario en Ingeniería Computacional
y Sistemas Inteligentes

Konputazio Zientziak eta Adimen Artifiziala Saila

Departamento de Ciencias de la Computación e Inteligencia Artificial

Master Tesia
Tesis de Máster

Evolutionary Computation in Hierarchical Model Discovery

David Revillas Rojo

Zuzendaritza
Dirección

Roberto Santana

Intelligent Systems Group

Department of Computer Science and Artificial Intelligence,

University of the Basque Country UPV/EHU

eman ta zabal zazu
Universidad del País Vasco Euskal Herriko Unibertsitatea

INFORMATIKA FAKULTATEA
FACULTAD DE INFORMÁTICA

Konputazio Ingeniaritza eta Sistema Adimentsuak Unibertsitate Masterra
Máster Universitario en Ingeniería Computacional y Sistemas Inteligentes

Master Degree Thesis:
Evolutionary Computation in Hierarchical Model Discovery

David Revillas Rojo

Advisors: Roberto Santana
Intelligent Systems Group,
Department of Computer Science and Artificial Intelligence,
University of the Basque Country UPV/EHU,
Paseo Manuel de Lardizabal, 1
Donostia, 20018 Gipuzkoa, Spain
drevillas@pm.me

Abstract. Despite its continuous growth, probabilistic programming is still a great unknown among scientists, specially those whose research areas involve sampling distributions, statistical modeling or statistical inference. This Master Thesis provides, on one hand, a novel procedure to learn and construct probabilistic programs that serve to model and sample probabilistic distributions. These probabilistic programs are based on grammatical rules through the potential given by evolutionary algorithms, concretely, the genetic programming approach. This technique provides a reliable back-end methodology that has served us to evolve a wide variety of program specifications and leading us, in a final step, to an optimal set of operations between distributions. These are visualized as a hierarchy, able to represent accurately any 1-dimensional tensor. On the other hand, the implemented framework offers the possibility of improving these models by calculating the best set of parameters for these learned models, with numerical optimization or distribution approximation methods, such as Markov Chain Monte Carlo techniques.

Keywords: Evolutionary Algorithms, Probabilistic Programming, Genetic Programming

Acknowledgments

I am not particularly eloquent when it comes to thanking people for the effort, desire and trust they have placed in me during this last stage. However, I believe that not only these people who have been accompanying me during these years deserve recognition, since, at the end of the day, all the people I have met along the way have been molding me and have made me what I am.

First and foremost, I would like to thank my parents, Elena and Guillermo, my brother Pablo, my grandmother and my uncle for giving me so much support, affection and stability throughout my life. It is clear that without their constant help I would not have reached the place where I am today.

On the other hand, I can't help but remember all the teachers from my school years, each one contributing their bit in my education, such as Cristina, who taught me to love mathematics as I do today; Roberto, that inveterate philosopher who had no trouble showing us with just how things are not what they seem to be; or my good old friend Aitor, who could be looking after the sheep or accompanying us to play a festival on the other side of the country.

To all those friends who have been passing by along the way, all those who have stayed and to the *Happy Gunners*, for the unforgettable moments that made an overwhelming week be forgotten with music, darts and beer in our usual place, to my partner Elena for that constant smile and illusion in me and also to the geeks of Iñaki, Borja and Unai for the scientific talks in the bars of Donosti.

And of course, thanks to my thesis director, Roberto, a mastodon in constant multitasking mode, for all the advice, help, motivation, opinions and effort deposited in me all this time, without which, no doubt, this would not have gone ahead.

How can computers learn to solve problems without being explicitly programmed? In other words, how can computers be made to do what needs to be done, without being told exactly how to do it?

— Arthur Samuel, 1959

Table of Contents

PART III
State of the art

PART IV
Representing probabilistic programs

PART V
Experimental framework

PART VI
Conclusions

List of Figures

List of Tables

List of Algorithms

Part I

Introduction

1 Introduction

Since the birth of Artificial Intelligence (AI) and more specifically, of Machine Learning (ML), one of the main objectives pursued by these techniques has been to analyze the existing data in order to discover hidden patterns or be able to predict the behavior of the systems or precess that generates the data. When modeling, it is often necessary to learn both the parameters of the models and their structure.

The concept of Probabilistic Programming may be new to many scientists and researchers. In this work, the meaning given to Probabilistic Programming refers to the area related to programming languages, this is, "usual" programs with the ability of sampling values at random from given distributions and the ability to condition values of variables in the presence of observed data [21]. The main purpose of this kind of programs is to specify a probability distribution. It also offers practitioners, the opportunity to model the data with a probabilistic approach, without the need for enough experience in probability theory or ML. An example of a simple probabilistic program can be seen in Algorithm 1, where tossing two coins modeled using two Bernoulli distributions and assigning the outcomes to check Boolean variables c_1 and c_2.

Algorithm 1 Coin toss probabilistic program.

1: bool c_1, c_2
2: float x_1
3: $x_1 \leftarrow Beta(2, 3)$
4: $c_1 \leftarrow Bernoulli(x_1)$
5: $c_2 \leftarrow Bernoulli(0.5)$
6: **return** c_1, c_2

However, the term Probabilistic Programming is given a different meaning in some research areas. In Appendix A, the reader can find a summary of areas where Probabilistic Programming is given a different meaning and the relationship with the type of approaches we focus on.

In this thesis, we present an automatic framework for the creation of generative models under the rules of Probabilistic Programming and Genetic Pro-

gramming (GP). Since such a program (or *individual* in GP terms) represents a generative model procedurally, our goal is to induce the code of these programs, as they are executed repeatedly, to be able to represent the given observed values. For this purpose, a similarity measure has been established, using summary statistics between the generated samples and the observed samples. All the work has been validated with different use cases, including a simple experiment of local modeling of precipitation in the Basque Country.

This document is made up of 5 parts. In Part 1, an overview of the problem to be addressed is given. In Part 2, the two fundamental pillars on which this work is based are introduced and detailed: Probabilistic programming and GP. Part 3 consists of an analysis of the state of the art and how automatic programming is being used today in different domains, such as ML. In Part 4, the approach developed in this work is described as well as the proposed grammar, types and functions. Part 5 describes in detail the configuration, evaluation and execution of the experiments carried out, as well as the results obtained. Finally, Part 6 presents the conclusions of the study.

Part II

Background

2 Probabilistic programs and Probabilistic Programming Languages

2.1 Definition of a probabilistic program

Probabilistic programming offers a constructivist way of describing probabilistic models, represented as common programs. They are written in any programming language like C, Python or Java, having the special ability to draw values at random from the specified distribution and also, the ability to condition values of variables in a program through observed data. These are not intended to be executed as a normal piece of code, but to implicitly specify a probability distribution. Despite of being able to represent probabilistic graphical models [27] which use graphs to specify conditional dependencies between random variables, this work does not contemplate that use.

Among others, one can find Probabilistic Programming Languages (PPLs) such as Church [19], one of the first probabilistic programming languages capable of representing any computable probability function. Venture [36], an interactive virtual machine designed for a general-purpose use, using a higher-order probabilistic language descended from Lisp. PyMC3 [49], an open source probabilistic programming framework written in Python, allowing model specification directly in Python code. Edward [54], a Turing-complete probabilistic programming language integrated into TensorFlow, which gives support to modelling neural networks and enables distributed training and Graphics Processing Unit (GPU) integration. Its operations are based on registering all random variables symbolically and not on execution, which has a considerable impact on memory usage. WebPPL [20], a lightweight successors to Church for Clojure and JavaScript. However, the language used in the present work is Pyro [7], as explained later.

Algorithm 1 shows a very simple probabilistic program where operators are only those of sampling from distributions. However, the probabilistic programming paradigm also considers *if-else* conditionals and loopy programs, as Algorithm 2 shows. This program will return `1` when the while loop is executed an even number of times and `0` when it is executed an odd number of times.

Algorithm 2 Loopy probabilistic program.

```
1: bool b, c
2: b ← 1
3: c ← Bernoulli(0.5)
4: while c do
5:     b ←!b
6:     c ← Bernoulli(0.5)
7: end while
8: return b
```

Thus, the expected value of `b` returned by the program can be seen as the probability that `b` is `1`, which is equal to the probability that the while loop executes an even number of times,

$$\mathbb{E}[\mathtt{b}] = p(\mathtt{b} = 1) = p(\textit{even loops})$$

For instance, the probability that the loops executes 0 times is given by the *Bernoulli* distribution in line 3 as

$$p(\textit{0 loops}) = p(c = 1) = 0.5$$

In the same way, the probability that the loop executes 2 times is obtained when, firstly, $c = 1$ in line 3, then $c = 1$ in line 6 the first time entered the loop and $c = 0$ the second time, so

$$p(\textit{2 loops}) = p(c = 1) \cdot p(c = 1) \cdot p(c = 0) = 0.5^3$$

Summing up for all even number of executions, the expected value that the loop executes an even number of times is obtained as:

$$\mathbb{E}[\mathtt{b}] = 0.5 + 0.5^3 + 0.5^5 + ... = \frac{2}{3}$$

2.2 Inference in Probabilistic Programming Languages

Apart from returning the expected value from the program, it is also possible to calculate the probability that the program terminates in a particular state. The expected value thus returned is the Probability Density Function (PDF) of the distribution of output states, also called posterior distribution. This is known as probabilistic inference. It is important to mention that exact inference is undecidable for programs with unbounded domains: exact inference is #P-complete [48].

A variety of inference techniques have been implemented in the probabilistic programming systems. These can be classified as:

– Static inference: the approach consists on compiling the program to a probabilistic graphical model and then, performing inference using algorithms such as belief propagation and its variants [43].

– Dynamic inference: another approach is to execute the program several times using sampling to execute probabilistic statements, observe the values of the desired variables on valid runs and compute statistics on the observed values to infer an approximation to the desired distribution.

Algorithm 1 shows a program that contains only three latent variables, x_1, c_1 and c_2. Each execution of a probabilistic program produces a unique execution trace. In turn, an execution trace is a mapping of random choices to their specific values. It completely defines the execution of the probabilistic program. This implies an infinite number of possible traces in which, at interpretation time, there is a branch at every random procedure. Given the execution trace, a probabilistic program becomes deterministic and its probability can be defined. An example of the execution trace for Algorithm 1 could be $x_1 = 0.69$, $c_1 = 0$ and $c_2 = 1$. As the authors of [59] propose, one can define the probability of an execution trace as

$$\tilde{p}(\boldsymbol{y}, \boldsymbol{x}) \equiv \prod_{n=1}^{N} p(y_n|\theta_{t_n}, \boldsymbol{x}_n)\tilde{p}(\boldsymbol{x}_n|\boldsymbol{x}_{n-1})$$

where $p(y_n|\theta_{t_n}, \boldsymbol{x}_n)$ is the likelihood of the observed output y_n where t_n is a random procedure (i.e., *Gamma*, *Poisson*, etc.), θ_{t_n} is its argument (possibly multidimensional), and $\boldsymbol{x}_n$ is the set of all random procedure application results computed before the likelihood of observation y_n is evaluated. $\sim$ denotes distributions which can only sample.

2.2.1 Exact inference

As mentioned above, exact inference is undecidable for programs with unbounded domains, i.e., the computational cost would be unaffordable. However, it can be useful in bounded problems or even to illustrate this type of inference. One can find different types of exact inference [11] but the illustrative ones are explained briefly below.

Forward reasoning This method can be used to construct a tree whose leaves and intermediate nodes contain the possible actions or values of the domain associated with a probability. For each level, a forward reasoning process is performed creating a child for each possible output, generating a unique world for each path to the leaves. Figure 1 shows an example. The probability of such a path is given by the product of the probabilities associated to the decisions made. However, this method needs to enumerate all possible alternatives, which in practice would be intractable for complex problems.

Backward reasoning This is a widely used inference strategy, and is based on obtaining the observations or evidence from a query, represented in an appropriate data structure, and using backward reasoning calculate the probability of that structure.

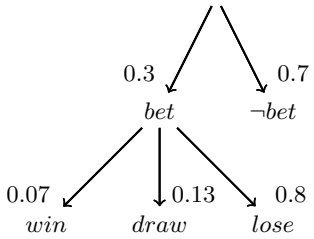


Figure 1: Forward reasoning illustrative example. The first level indicates the probability that a player bets, while in the second level, if he wins the bet, he loses it or recovers his investment.

2.2.2 Approximate inference

To circumvent all the difficulties of exact inference in most real-world problems, approximate inference arises. Formally, these algorithms can be distinguished into two categories:

Sampled-based inference This method consists of randomly sampling a large number of possible events, from which the query is estimated. The most popular method of all are the Markov Chain Monte Carlo (MCMC) algorithms (Metropolis Hastings [9] sampling, Gibbs sampling [15, 16], Hamiltonian Monte Carlo [6] sampling), which instead of generating all decisions from the beginning, generate sequences of samples by applying random modifications to the previous sequences. They can provide accurate estimates, but they are usually slow for programs with a complex structure.

Variational inference Although faster than the MCMC methods, these methods calculate an approximate estimate [31]. The idea is to use a small family of functions (usually parameterized so as to optimize the procedures) instead of using the entire function space and obtain the posterior. A key issue in these methods is how to measure the distance between the approximate posterior and the observed posterior. In practice, the Kullback-Leibler (KL) divergence is used.

2.3 Pyro

The selected PPL to work with in the project was Pyro [7]. Pyro is a flexible and scalable deep probabilistic programming library built on PyTorch. In turn, PyTorch provides GPU based tensor computation, unlike the well known NumPy library. It is also used in Deep Learning (DL) research.

Pyro was designed to be: universal, able to represent any computable distribution; scalable, able to handle large data sets and high-dimensional models common in AI research; flexible, to ensure researchers a quick and easy implementation of the ideas; minimal, sharing most of its syntax and semantics with existing languages, like in this case, Python.

However, Pyro is not the only language intended for these purposes. PyMC3 [49] offers similar functionalities with Aesara as computational backend (formerly Theano). Google has also its own language under TensorFlow ecosystem, called TensorFlow Probability [1], which provides integration of probabilistic methods with deep networks and scalability to large datasets and models via hardware acceleration and distributed computation.

```
import pyro

a = pyro.distributions.Beta(0.4, 1)
b = pyro.distributions.Poisson(3)

p = a.sample([5]) / b.sample([5])
y = pyro.distributions.Bernoulli(p)
```

Algorithm 1.1: Program example written in Pyro. Latent variables `a` and `b` are represented by a Beta and Poisson distribution, respectively. Then, a simple arithmetic operation is performed between them, after sampling 5 value each. Finally, the random variable `y`, parameterized by the random tensor `p` of 5 elements, simulates the generative process returning a binary vector of also 5 elements.

2.4 Graph representation of Probabilistic Programming Languages

As seen, Algorithm 2 presents a pseudocode of a simple probabilistic program and Algorithm 1.1 under the Pyro programming language. However, a prior step to describing these programs in a concrete language is to sketch such probabilistic models in a mathematical notation. Suppose the following hierarchical model:

$$\begin{aligned}
\mu &\sim \mathcal{N}(0, 1) \\
\lambda &\sim Poisson(3) \\
y &\sim Exp(\lambda + \epsilon) \\
z &\sim \mathcal{N}(\mu, y)
\end{aligned}$$

where λ, μ and y_1 are hyper-parameters, i.e. parameters that influence other parameters. The use of a very small value ϵ is necessary for the model to be consistent, since the Poisson distribution can sample zeros, while the exponential distribution requires arguments strictly greater than 0. In this model, a normal distribution and a Poisson distribution are used to model the hyperprior exponential distribution and this in turn, the final normal distribution.

However, there is another way of expressing hierarchical models by means of a graphical representation, as shown in Figure 2.

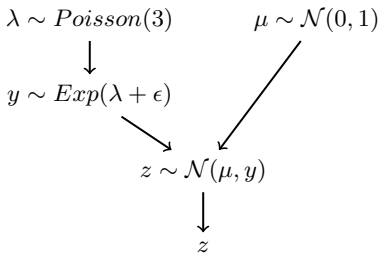


Figure 2: Graphical representation of a hierarchical model.

2.5 Probability distributions for Probabilistic Programming Languages

It is important to note, that although a probabilistic program is used to represent distributions, it also needs to use distributions internally. In fact, the values generated by one distribution are used as parameters of another. Such a process can be organized by specifying what kind of values a distribution accepts as parameters, and what kind of values it generates. Since there are a large number of distributions, both discrete and continuous, this project has considered only those shown in Table 1.

#	Distribution	Support	D. parameters	Pyro parameters
1	Bernoulli	$k \in \{0, 1\}$	$0 \leq p \leq 1$	p: `probs`
2	Binomial	$k \in \mathbb{N}_0$	$n \in \mathbb{N}_0,\ p \in [0, 1]$	n: `total_count`, p: `probs`
3	Poisson	$k \in \mathbb{N}_0$	$\lambda \in (0, +\infty)$	p: `rate`
4	Beta	$x \in [0, 1]$	$\alpha, \beta > 0$	α: `concentration1`, β: `concentration2`
5	Chi-square	$x \in [0, +\infty)$	$k \in \mathbb{N}$	k: `df`
6	Exponential	$x \in [0, +\infty)$	$\lambda > 0$	λ: `rate`
7	Normal	$x \in \mathbb{R}$	$\mu \in \mathbb{R},\ \sigma > 0$	μ: `loc`, σ: `scale`

Table 1: Distributions used to create probabilistic programs and their constraints.

Each of the distributions has its own nature. Therefore, it is possible to construct a graph showing the dependencies between distribution parameters, as shown in Figure 3, that will provide us a way to generate a probabilistic program where inputs and outputs of the distribution are consistent. This will be further discussed in the following chapters.

However, in order not to restrict ourselves to just the dependencies shown in Figure 3, the proposed method also adds functions to enable filling parameters between every distribution, as discussed in the next chapters.

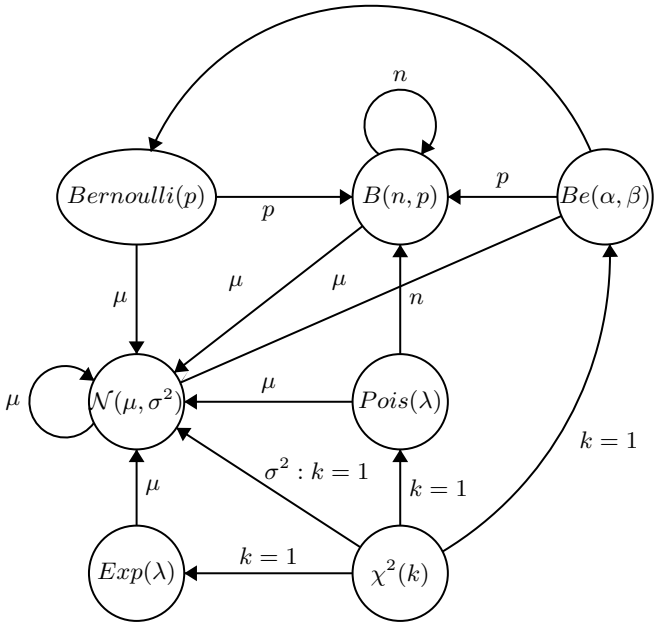


Figure 3: Dependencies between distributions. Each node of the network represents a distribution. Each edge represents a relationship with another distribution. This means that the target distribution allows its parameters to be generated by the source distribution. For example, when $k = 1$ in a χ^2 distribution, it can be used to generate the σ^2 of a normal distribution or the λ of a Poisson distribution.

3 Genetic Algorithms and Genetic Programming

3.1 Definition

Genetic Algorithms (GAs), as a subset of Evolutionary Algorithms (EAs), represent a set of population-based metaheuristic techniques based on mimicking the evolution, using the Darwinian principle of reproduction and survival of the fittest. Its beginnings date back to the works of Holland [24] in the 70's and later in the 80's with his PhD students Koza [28] and Goldberg [17, 18].

The generic GA evolves a population of individuals, with an associated fitness value. Each iteration of the algorithm is called a generation and each individual, also known as *chromosome*, represents a possible solution to a given problem. Thus, the GA tries to find the best or at least, a very good solution to the problem by genetically breeding the population over a series of generations. This is achieved, firstly, by designing the representation scheme of the chromosome. This representation can play a crucial role in the optimization process in terms of computation time, memory space and reachability of the global optimum. In that scheme, each location (a gene) is associated with a particular variable of the problem, denoting the value of a particular variable (allele). This is usually composed by binary values, strings or integers.

As mentioned, each individual in the population has a fitness value given by an objective function, also user-defined. This measure controls how well it fits to the problem in order to select the best candidates to breed the next generations.

Another important aspect of the evolution process are the choice of the reproduction, mutation and selection operators, and the parameters for controlling the algorithm itself: population size and the maximum number of generations. Finally, a termination criterion is required for deciding when to stop the evolution process.

Once the previous concepts have been defined, the pseudocode of a general GA is described in Algorithm 3.

Selection Most of the EAs uses a fitness criterion to probabilistically select individuals. That is, better individuals are more likely to have more descendants. There are some methods to perform this selection: roulette wheel selection, rank selection, elitism, tournament selection, etc. Any mechanisms can be used in the selection. However, it is worth to mention the *selection pressure*: a system with a strong selection pressure will favor the fittest individuals, promoting a superindividual, while a system with a weak pressure will not be so discriminating at the time of selecting solutions.

In this project, tournament selection is used. In tournament selection a number of individuals are chosen at random from the population and then they are compared with each other so the best one is chosen as a parent. In the crossover, two parents are needed, so two selection tournaments are made. This method automatically rescales fitness in order to make the selection pressure constant.

Algorithm 3 Genetic Algorithm.

```
1: Randomly create an initial poppulation P'_0
2: k ← 0
3: repeat
4:     Assign a fitness value to each individual
5:     Select n individuals from P'_k
6:     k ← k + 1
7:     for n/2 times do
8:         Randomly choose two individuals from the selected ones
9:         Cross the individuals with probability p_c
10:        Mutate the obtained individuals with probability p_m
11:        Introduce the two new individuals in population P'_k
12:    end for
13: until stopping criteria
14: return best solution
```

Tournament selection also amplifies small differences between solutions, choosing a solution even if it is only marginally superior to the others.

Crossover Crossover, also called recombination, is another genetic operator used to combine genetic information of two individuals to make a third one. There exist many methods to recombine the information of two chromosomes. The single-point crossover and the k-points crossover are the best known methods for chromosomes represented by a bit array. However, there are several more variants of crossover. Despite the technique used, it is important that the applied one ensures legal solutions. For example, combining two permutations with a single point crossover could lead to a solution that is not a permutation.

Mutation Mutation is an operator in charge of maintaining the genetic diversity of the population into the next generations, avoiding getting stuck in a local minima by preventing the population of chromosomes from becoming too similar to each other. In the simplest cases, it works by flipping a bit at random. However, it also needs to ensure the validity of a solution.

3.2 Genetic Programming

Any computer program can be graphically depicted as a tree with ordered branches. However, GP is not typically used to evolve programs in the familiar Turing-complete languages[1] humans normally use for software development. In fact, GP is not Turing-complete [53, 60]. It is instead more common to evolve programs (or expressions or formulae) in a more constrained and often domain-specific language.

[1] A Turing-complete language refers to an environment in which a program able to find a solution can be written, with no guarantees regarding runtime or memory.

The overview presented earlier shows a generic approach to any genetic algorithm. As subfield of it, in GP an evolution of computer programs is performed. Generation by generation, GP stochastically transforms populations of programs into new ones. In this case, the fitness value is computed by running the program and comparing the results of it to some ideal.

The search space in GP is the space of all possible computer programs composed of functions and terminals appropiate to the problem domain. When applying GP, there are some considerations to made before:

1. The set of terminals: these can be viewed as the inputs to the as-yet-undiscovered computer program.
2. The set of primitive functions: used to generate the mathematical expression that attempts to fit the given finite sample of data.
3. The fitness measure.
4. The parameters for controlling the execution.
5. The stopping criterion.

A general pseudocode of a GP is presented in Algorithm 4.

Algorithm 4 Genetic Programing.

```
1: Randomly create an initial poppulation of programs from the available primitives
2: repeat
3:     Execute each program and compute its fitness
4:     Select one or two program(s) according to a probability based on the fitness
5:     Create new program(s) by applying genetic operations according to a predefined
   probability
6: until stopping criteria are satisfied
7: return best solution
```

3.3 Representation

In GP, programs are usually expressed as *syntax trees* rather than as lines of code. The variables and constants in the program are leaves of the tree, and they are called *terminals*, whilst the arithmetic operations are internal nodes called *functions*. The sets of allowed functions and terminals together form the *primitive set* of a GP system.

Consider the terminal set T and the function set F defined below:

$$T = \{x, y, 3, 4\}$$
$$F = \{+, -, /, *, max, min, log, pow\}$$

A simple example program is shown in Figure 4.

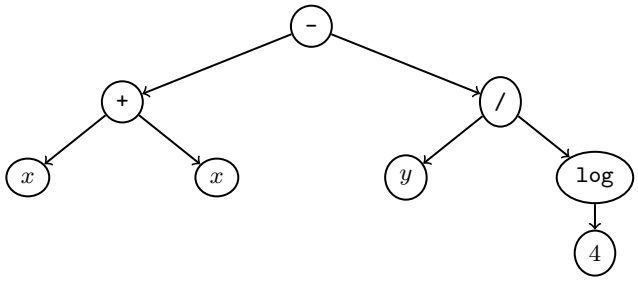


Figure 4: Genetic Programming syntax tree representing `(x+x) - (y / log(4))`. `x`, `y` and `4` represent the terminal nodes, while `-`, `+`, `/` and `log`, the functions.

3.3.1 Terminal set

The terminal set may consist of:

- The program's external inputs, typically taking the form of named variables.
- Functions with no arguments, such as the `rand()` function which returns random numbers.
- Constants, which can be pre-specified or randomly generated.

However, using a primitive such as `rand` can cause the behaviour of an individual program to vary every time. It is preferable to have a set of fixed random constants that are generated as part of the process of initializing the population. This is known as an *ephemeral random constant*. Every time this terminal is chosen in the construction of a tree, a different random value is generated and will remain fixed for the rest of the run.

3.3.2 Function set

The function set is typically driven by the nature of the problem. This set may consist of simple arithmetic functions like `+`, `-`, `*` and `/`, but it is also possible to include any other functions encountered in computer programs.

3.3.3 Closure

An important property required for GP to work properly is the *closure* [29], which can be defined as *type consistency* and *evaluation safety*.

Type consistency Required because subtree crossover can mix nodes arbitrarily. Thus, it is necessary that any subtree can be used in any of the argument positions for every function in the function set. This forces all the functions to be type consistent, i.e., they all return values of the same type. It is worth to mention that the system developed in this project is, in fact, strongly typed, satisfying this first property.

Evaluation safety The other component of closure is required because many commonly used functions can fail at run time. An evolved expression might, for example, divide by 0. This is typically dealt with by modifying the normal behaviour of primitives. It is common to use protected versions of numeric functions that can otherwise throw exceptions, such as division, logarithm, exponential and square root. If a problem is spotted then some default value is returned.

An alternative to protected functions is to control exceptions at runtime and considerably reduce the fitness of programs that generate such errors. This can also lead to a situation where many individuals generate invalid expressions with a very high probability, which can cause the entire population to have the same very poor fitness.

3.4 Genetic Programming operators

3.4.1 Individual generation

There are some approaches to generating individuals, but here only two of the simplest ones are presented: the *full* and the *grow* methods. In both, the *full* and *grow*, the generated individuals do not exceed a user specified maximum depth. That is, the depth of its deepest leaf. However, there are other methods like *ramped half-and-half* [29] that combine both the *full* and the *grow* methods or the *ramped uniform initialization* [32], that allows the user to specify a range of initial tree sizes.

The *full* method In this method, nodes are taken at random from the primitive set until the maximum tree depth is reached. Figure 5 shows a series of steps of the construction of a full tree of depth 2. Although this method generates trees where all the leaves are at the same depth, this does not necessarily mean that all initial trees will have an identical number of nodes.

The *grow* method This method allows for the creation of trees of more varied sizes and shapes. Nodes are selected from the whole primitive set until the depth limit is reached. Once the depth limit is reached only terminals may be chosen. Figure 6 illustrates this process.

3.4.2 Selection

The most commonly employed method for selecting individuals in GP is tournament selection, as explained earlier in Section 3.1.

3.4.3 Recombination and mutation

In order to cross individuals in GP, there is a need to design sensible ways of "combining" trees. As explained in the generation of individuals, one can not simply apply a one-point crossover. Instead, a *subtree crossover* is applied. Given

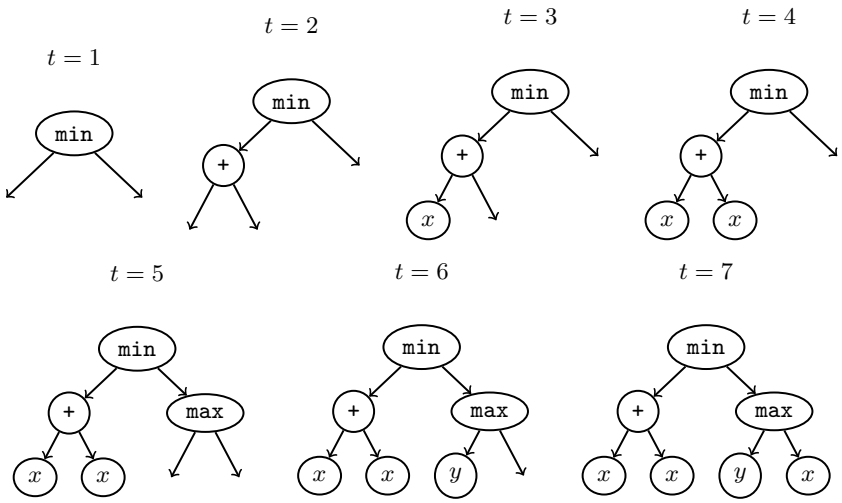


Figure 5: Creation of a full tree having maximum depth 2 using the *full* method, with terminal set T and function set F defined earlier, (t = time).

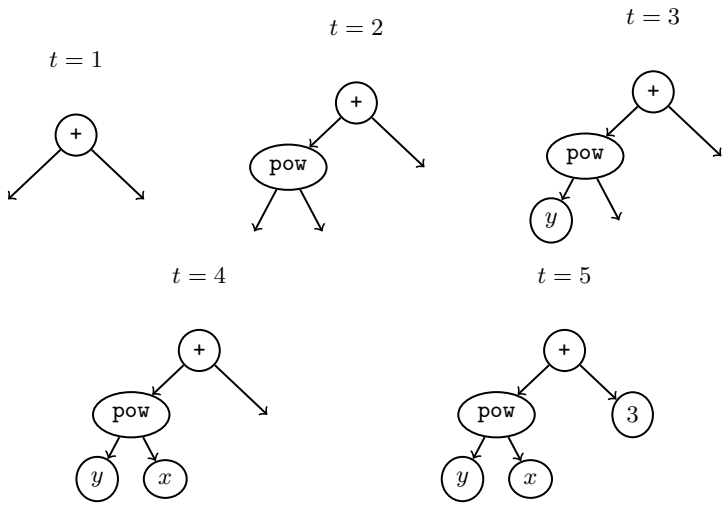


Figure 6: Creation of a five node tree using the *grow* initialisation method with a maximum depth of 2, using terminal set T and function set F defined earlier, (t = time).

two parents, this method randomly selects a *crossover* point (a node) in each parent tree and then, it creates the offspring by replacing the subtree rooted at the crossover point in a *copy* of the first parent with a *copy* of the subtree at the crossover point in the second parent, as shown in Figure 7.

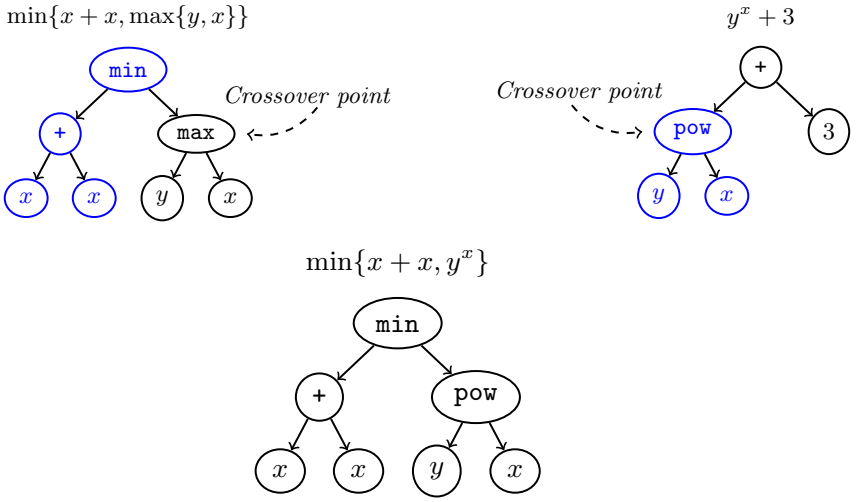


Figure 7: Example of subtree crossover. The blue-colored subtrees identify the genetic material shared by the parents in the offspring.

Regarding GP mutation, a mutation point is randomly selected in the tree and it substitutes the subtree rooted there with a randomly generated tree, as illustrated in Figure 8. When subtree mutation is applied, this involves the modification of exactly one subtree. Other types of mutation operators can be applied as explained in [45].

Some comments about the GP implementation A protected version of the genetic initialization operators `genFull()`, `genHalf()` and `genHalfAndHalf()` had to be implemented due to a possible bug in the DEAP library code. This problem arose when a new individual was created and the tree structure attempted to add a primitive type to a node when it should actually add a terminal. This is explained in the GitHub repository issues tracker[2].

In any case, it should be made clear that the population initialization mechanism used was the previously mentioned *ramped half-and-half*, which combines the *full* and *grow* methods. For the case of mutations, only the *full* method has been used.

About the recombination, the `cxOnePoint()` crossover and `mutUniform()` are applied.

[2] https://github.com/DEAP/deap/issues/237#issuecomment-508087233

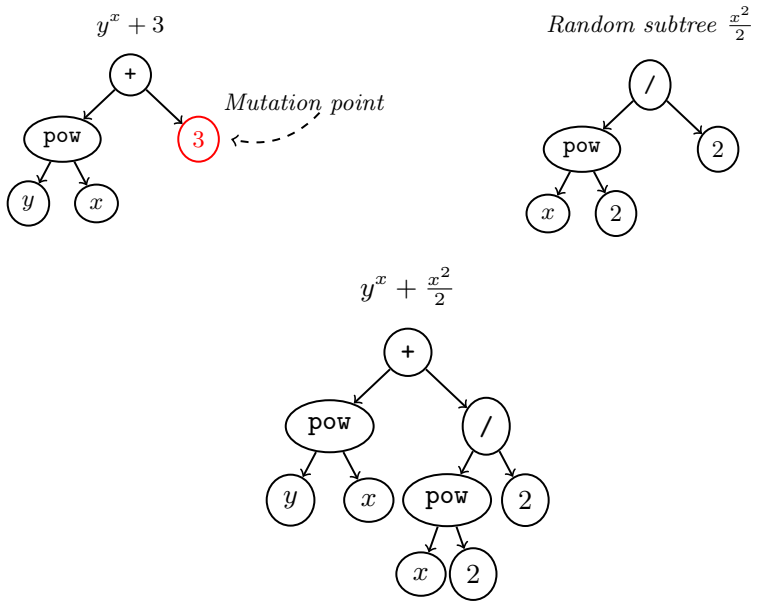


Figure 8: Example of subtree mutation. The red-colored subtree shows the mutation point which will be replaced entirely by a randomly generated subtree.

Part III

State of the art

4 Related work

4.1 Automatic programming

As mentioned in previous chapters, the objective is to learn programs capable of generalizing observed data through the synthesis of concrete code and specifications. This is known as Inductive Programming (IP) and addresses the problem of learning programs from incomplete specifications, such as input/output examples, traces or constraints, which is a topic of interest in AI research since the 1960s. There exist many approaches to program synthesis distributed over different communities, like inductive logic programming [14, 25, 35, 39], evolutionary programming [29, 42] and functional programming [13, 26, 50].

In the context of probabilistic programming, current approaches to learning these programs do, basically, know in advance the hierarchical model or the data distribution, and then proceed with the parameter learning phase. However, the assumed hierarchy specification could be inaccurate or incomplete.

The approach proposed in [22] turns out to be very interesting and very similar to the objectives set out in our work. As in this thesis, the goal of the authors is to determine the appropriate structure of a model based on observed data, so that this modeling is performed by non-experts. However, their proposal contemplates what are known as matrix decompositions: common modeling assumptions, which are expressed by a class of probabilistic models. In such a model, the component matrices are first sampled independently from a small set of factors and then combined using simple algebraic operations. This is precisely what the authors of [22] intend to exploit, the structural composition, to evaluate, infer the decomposition matrix, and automatically search the structure space.

Similar to the approach followed in this thesis is also the work of Perov [44], in which the author aims to develop a technique for models, represented as programs, to be able to generalize from data.

In the work of [34], authors propose a solution based on combinatory logic to represent complex programs through simpler subprograms. To do so, they developed a new form of combinators logic, motivated by the need to transform the program defined in lambda calculus into a variable-free representation. One of the major problems of lambda calculus is, as they point out, the long-range dependencies between where a variable is bound (λx) and the places where it is

used (x). After defining a proper grammar and its types, the goal is to define a probabilistic model, i.e., a distribution over *combinators* (binary trees whose leaves ares primitive combinators) for each type. Finally, authors of [34] perform inference to find the posterior distribution of the latent programs as a function of the training cases, all using the MCMC inference approximation. Thus the introduced algorithm manages to evolve a program dedicated to perform an arithmetic task and another program focused on text editing.

In fact, the authors of [8] published 4 years earlier a similar idea, rejecting the use of lambda calculus in favor of combinatory logic, pushing the idea of developing these combinatorial expressions as program representations for genetic programming, since it made evolution possible using simple genetic operators. In addition, they also suggested that the use of a functional language facilitated the search more than an imperative language.

Mansinghka's PhD thesis [37] introduces techniques within the field of probabilistic computing, where one can use such tasks for the specification of generative models for example, as well as generalizing and parallelizing sampling algorithms such as MCMC. Among other things, the thesis also introduces digital stochastic circuits capable of modeling probabilistic algebra as Boolean circuits do with Boolean algebra, including how to implement these circuits massively to process these samplings and efficiently run such algorithms as MCMC with thousands of variables, even in real time.

4.2 Genetic programming for Machine Learning

Some other works have described applications of other ML problems. We discuss some of these applications here. Krawiec's work [30] aims to help ML classifiers improve their performance while maintaining their readability by humans. To this end, they present a procedure based on GP for constructing new features. This process is carried out by deriving new features from the original ones and searching for a suboptimal set of them. In this case, each individual in the population represents a set of features defined as LISP expressions. That is to say, they obtain a symbolic and understandable representation once the evolution is finished.

Another paper, concerning classification problem solving, is [58]. In this case, the researchers interpret classification problems as optimization problems and assume that each instance of the classification problem is an optimization problem and the solution is found by means of heuristics. The contribution of this research is based on the development of a set of genetic operators suitable for this task and new algorithmic concepts, such as Segregative Genetic Algorithm (SEGA) [2] and its further development Self Adaptive Segregative Genetic Algorithm (SASEGASA) [3].

These ML tasks, as pointed out, often include feature selection and engineering, missing value imputation, model selection, training and validation... In short, time-consuming tasks. For this, among other reasons, the concept of Automated Machine Learning (AutoML) arose. The high degree of automation allows non-expert users to apply models and techniques without much effort.

Some developments around this idea may be Tree-based Pipeline Optimization Tool (TPOT) [41], that automatically designs and optimizes machine learning pipelines for a given problem domain without any need for human intervention, using GP.

Conversely, statistical ML concepts have also been introduced into the evolutionary world. The survey [4] presents a wide variety of elements that aim to improve the evolution of GP systems, such as model selection (validation methods, analytical methods and feature selection), fitness evaluation (regularization, fitness functions or sampling methods) or the search for operators and selection schemes.

Another concept that generates discussion within the ML community is the explainability of these models. Within a classification task for example, it may be easy to explain why a k Nearest Neighbor model has made that class prediction for a given instance, or even a Decision Tree. However, there are models known as *black-box*, where deriving and explanation for the model behaviour becomes somewhat complicated, as in Neural Networks. The work described [12], encompassed within the area of Explainable AI (XAI), proposes the use of a multi-objective GP method to address these issues, whose objectives are to propose a simple structure in tree form and explainable, which reconstructs the prediction scheme of a certain model, maximizing in turn the reconstruction ability and minimizing the complexity of such trees. According to the authors, this procedure is applicable to any black-box classifier and without making any a priori assumptions about them.

In summary, the current uses of GP in ML are numerous and varied. In this related work section, we have reviewed only a representative number of works.

Part IV

Representing probabilistic programs

5 Specifying probabilistic programs with a grammar

5.1 Proposed grammar

As mentioned in previous sections, the GP trees developed in this work use *strongly typed* GP, where every terminal has a type and every function has types for each of its arguments and a type for its return values. The process that generates the initial, random expressions, and all the genetic operators are implemented so as to ensure that they do not violate the type system's constraints. For example, as the reader may know, a Normal distribution takes two input arguments: the mean μ and the standard deviation σ. Each of those arguments has its own restrictions: $\mu, \sigma \in \mathbb{R}$ and $\sigma > 0$. The types defined for the GP program will be related to the parameters that define a distribution, and the output of a distribution. Thus, the implementation of a *strongly typed* GP system ensures that any value passed to σ argument will be strictly positive, despite being a value or a function output.

A natural way to express these constraints is via *grammars*, as expressed in Figure 9. Each line in the grammar is a *production rule*. Elements that cannot be rewritten are known as the *terminals*[3], while symbols that appear on the left side of a rule are known as *non-terminal*.

A GP grammar is typically used to ensure that the initial population is made up of "grammatically correct" programs and to guide the operations of the genetic operators. Thus, with this system, crossover and mutation is restricted to only swapping subtrees derived from a common non-terminal symbol in the grammar. For instance, a subtree rooted by a $\langle tensor \rangle$ node could be only replaced by another also rooted by a $\langle tensor \rangle$.

Figure 10 shows an example of a derivation tree to grammatically represent the individual (model) `Normal(x + y, Exponential(z))`.

[3] Not to be confused with the terminals in the primitive set of a GP system.

⟨*tree*⟩ ::= $\mathcal{D}$ `Represents a Pyro distribution object`

⟨$\mathcal{D}$⟩ ::= ⟨*N01D*⟩ $x \in \{0, 1\}$
| ⟨*N0D*⟩ $x \in \mathbb{N}, x \geq 0$
| ⟨*R01D*⟩ $x \in \mathbb{R}, x \in [0, 1]$
| ⟨*RD*⟩ $x \in \mathbb{R}$
| ⟨*RP0D*⟩ $x \in \mathbb{R}, x \geq 0$

⟨*N01D*⟩ ::= '`Bernoulli(`' ⟨*R01T*⟩ '`)`' `Pyro's Bernoulli distribution`

⟨*N0D*⟩ ::= '`Binomial(`' ⟨*N0T*⟩ '`,`' ⟨*R01T*⟩ '`)`' `Pyro's Binomial distribution`
| '`Poisson(`' ⟨*RPT*⟩ '`)`' `Pyro's Poisson distribution`

⟨*R01D*⟩ ::= '`Beta(`' ⟨*RPT*⟩ '`,`' ⟨*RPT*⟩ '`)`' `Pyro's Beta distribution`

⟨*RD*⟩ ::= '`Normal(`' ⟨*tensor*⟩ '`,`' ⟨*RPT*⟩ '`)`' `Pyro's Normal distribution`

⟨*RP0D*⟩ ::= '`Exponential(`' ⟨*RPT*⟩ '`)`' `Pyro's Exponential distribution`
| '`Chi2(`' ⟨*NT*⟩ '`)`' `Pyro's Chi2 distribution`

⟨*R01T*⟩ ::= '`Tensor.abs(`' ⟨*tensor*⟩ '`).clip(0, 1)`' $x \in \mathbb{R}, x \in [0, 1]$

⟨*N0T*⟩ ::= '`Tensor.abs(`' ⟨*tensor*⟩ '`).round()`' $x \in \mathbb{N}, x \geq 0$

⟨*RPT*⟩ ::= '`Tensor.abs(`' ⟨*tensor*⟩ '`).clip(10e-7)`' $x \in \mathbb{R}, x > 0$

⟨*NT*⟩ ::= '`Tensor.abs(`' ⟨*tensor*⟩ '`).clip(1).round()`' $x \in \mathbb{N}, x > 0$

⟨*tensor*⟩ ::= '`Tensor(`' ⟨*sample*⟩ '`)`' `Converts Pyro's sample (or any sequence) to tensor`
| '`Tensor(`' ⟨*aop*⟩ '`)`' `Arithmetic operators`
| '`Tensor(`' ⟨*tensor*⟩ '`)`' `Another tensor`
| '`Tensor(`' ⟨*input*⟩ '`)`' `Input tensor`

⟨*sample*⟩ ::= '`sample(_,`' ⟨$\mathcal{D}$⟩ '`)`' `Pyro's sample`

⟨*aop*⟩ ::= ⟨*add*⟩ `Generic addition`
| ⟨*sub*⟩ `Generic substraction`
| ⟨*mul*⟩ `Generic product`
| ⟨*safediv*⟩ `Protected division`
| ⟨*safepow*⟩ `Protected exponentiation`

⟨*add*⟩ ::= ⟨*tensor*⟩ '`+`' ⟨*tensor*⟩

⟨*sub*⟩ ::= ⟨*tensor*⟩ '`-`' ⟨*tensor*⟩

⟨*mul*⟩ ::= ⟨*tensor*⟩ '`*`' ⟨*tensor*⟩

⟨*safediv*⟩ ::= '`safediv(`' ⟨*tensor*⟩ '`,`' ⟨*tensor*⟩ '`)`'

⟨*safepow*⟩ ::= '`safepow(`' ⟨*tensor*⟩ '`,`' ⟨*tensor*⟩ '`)`'

⟨*input*⟩ ::= x | y | z `Input values, i.e. list, tensor`

Figure 9: Grammar introduced to represent probabilistic programs in Pyro.

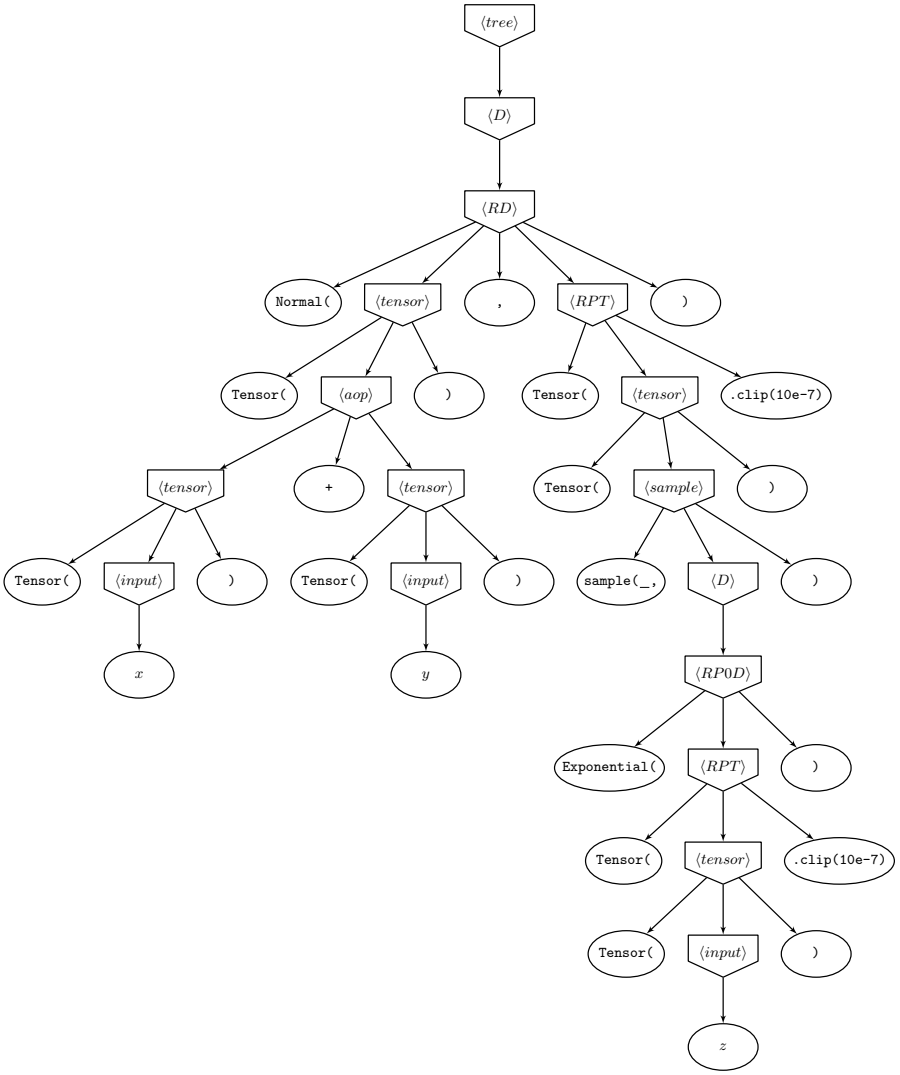


Figure 10: Example individual represented by the derivation tree for the program `Normal(x + y, Exponential(z))`.

5.1.1 Types implemented

Figures 11, 12 and tables 2, 3 summarize some of the characteristics of the introduced grammar.

First, it is necessary to characterize the output of each of the distributions, i.e., the type of tensor returned and the domain of values for the variables. It is important to remember that this distribution output is a sample of data. The

types shown in Table 2 are implemented in the grammar for this purpose. Figure 11 shows the corresponding class diagram.

Tensor type	Represents	Domain
`RealPositiveTensor`	`torch.Tensor`	$x, x \in \mathbb{R}, x \in (0, +\infty)$
`RealPositive0Tensor`	`torch.Tensor`	$x, x \in \mathbb{R}, x \in [0, +\infty)$
`Real01Tensor`	`torch.Tensor`	$x, x \in \mathbb{R}, x \in [0, 1]$
`NaturalTensor`	`torch.Tensor`	$x, x \in \mathbb{N}, x \in (0, +\infty)$
`Natural0Tensor`	`torch.Tensor`	$x, x \in \mathbb{N}, x \in [0, +\infty)$
`Natural01Tensor`	`torch.Tensor`	$x, x \in \mathbb{N}, x \in [0, 1]$

Table 2: Tensor types.

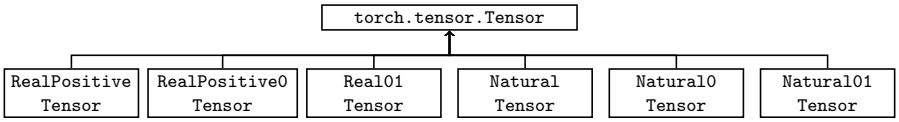


Figure 11: Class diagram showing tensor types defined in the implementation.

On the other hand, the type of distribution also had to be determined. For instance, a sample from a normal distribution will not have the same characteristics as the output of a Beta, since the former will have a support in $\mathbb{R}$ and the latter in $[0, 1]$. Therefore, in the same way, Table 3 and Figure 12 show the corresponding type hierarchy.

Distribution type	Represents	Domain
`RealDistribution`	`pyro.Distribution`	$x \in \mathbb{R}, x \in (-\infty, +\infty)$
`NaturalDistribution`	`pyro.Distribution`	$x \in \mathbb{N}, x \in (0, +\infty)$
`Natural0Distribution`	`NaturalDistribution`	$x \in \mathbb{N}, x \in [0, +\infty)$
`Natural01Distribution`	`NaturalDistribution`	$x \in \mathbb{N}, x \in [0, 1]$
`RealPositiveDistribution`	`RealDistribution`	$x \in \mathbb{R}, x \in (0, +\infty)$
`RealPositive0Distribution`	`RealDistribution`	$x \in \mathbb{R}, x \in [0, +\infty)$
`Real01Distribution`	`RealDistribution`	$x \in \mathbb{R}, x \in [0, 1]$

Table 3: Distribution types.

However, it is worth noting that these newly created types serve to create an ecosystem of types to implement the `PrimitiveSetTyped` of Distributed Evolutionary Algorithms in Python (DEAP). In terms of Python implementation, it is

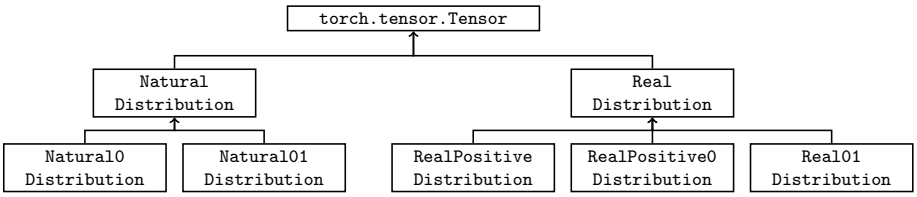


Figure 12: Class diagram showing distribution types defined in the implementation.

not necessary to develop any code regarding these types, it is enough to simply define the header of the class itself. This is useful for handling the output types of the functions we define, as well as the distributions and operations between them.

5.1.2 Function set

Some unary operators have been added to the function set. The function of the unary operators is to specify not only the output type, but also to transform the type of the tensor and its values. For example, by applying the function `toNaturalTensor` to an arbitrary tensor x, the system will transform it so that it satisfies that all its values $x_i \in \mathbb{N}$. This transformation in concrete, is performed according to $\lfloor \max(1, |x|) \rceil$. These unary operators are described in Table 4.

Operator		**Input**	**Output**
`toReal01Tensor`	Converts to $\mathbb{R} \in [0, 1]$ domain	`torch.Tensor`	`Real01Tensor`
`toRealPositiveTensor`	Converts to $\mathbb{R} \in (0, +\infty)$ domain	`torch.Tensor`	`RealPositiveTensor`
`toNaturalTensor`	Converts to $\mathbb{N} \in (0, +\infty)$ domain	`torch.Tensor`	`NaturalTensor`
`toNatural0Tensor`	Converts to $\mathbb{N} \in [0, +\infty)$ domain	`torch.Tensor`	`Natural0Tensor`

Table 4: Unary operators.

Similarly, basic arithmetic operations have been proposed as binary operators. These will receive a pair of arbitrary tensors and return a generic tensor. The grammar will then perform the appropriate conversions between tensor types to continue building a valid program. It is worth mentioning that protected versions of division have been implemented to avoid dividing by zero, and of exponentiation, to avoid a negative floating point power. They are described in Table 5.

It is perhaps worth commenting on the fact that basic functions such as `abs()` or rounding have not been implemented, since they are included in the implementation of the different variants of the conversion to tensors `to*Tensor`.

Operator		Input	Output
`add`	Python's standard addition	`[torch.Tensor, torch.Tensor]`	`torch.Tensor`
`sub`	Python's standard subtraction	`[torch.Tensor, torch.Tensor]`	`torch.Tensor`
`mul`	Python's standard product	`[torch.Tensor, torch.Tensor]`	`torch.Tensor`
`safeDiv`	Protected division	`[torch.Tensor, torch.Tensor]`	`torch.Tensor`
`safePow`	Protected exponentiation	`[torch.Tensor, torch.Tensor]`	`torch.Tensor`

Table 5: Binary operators.

Part V

Experimental framework

6 Problem definition

As discussed in previous chapters, the general objective of this work is to evolve, using GP, probabilistic programs able to sample complex distributions from which only a small sample of observed data points is available. To frame our analysis, we will use the formulation presented by Perov and Wood in their work [44] to represent the *program generation process*:

$$\pi(\mathcal{X}|\hat{\mathcal{X}})p(\hat{\mathcal{X}}|\mathcal{T},\theta)p(\mathcal{T}|\theta)p(\theta) \tag{6.1}$$

where:

- θ represents the input parameters of the program $\mathcal{T}$.
- $p(\theta)$ is the probability distribution of θ.
- $p(\mathcal{T}|\theta)$ is the distribution of $\mathcal{T}$ given θ.
- $\hat{\mathcal{X}}$ is the data generated by the probabilistic program $\mathcal{T}$ parameterized by θ.
- $\mathcal{X}$ is the observed data, e.g. from sensor readings, experiments, etc.
- $\pi(\mathcal{X}|\hat{\mathcal{X}})$ is a distance between the summary statistics computed between $\mathcal{X}$ and $\hat{\mathcal{X}}$.

The current implementation of GP approach a vector of 3 components to represent the parameters (θ) for each program $\mathcal{T}$. The parameter values were originally sampled from a uniform distribution, but this could be generalized assuming the existence of a latent variable β from which the inputs are generated and defining the conditional distribution of the input parameters given this latent variable, i.e., $p(\theta|\beta)$. For instance, instead of generating θ from the uniform distribution, we could generate it from $\mathcal{N}(\beta, 1)$, where in the current implementation β has three components and will generate three columns of parameters. In the general case, Equation 6 would be modified as follows:

$$\pi(\mathcal{X}|\hat{\mathcal{X}})p(\hat{\mathcal{X}}|\mathcal{T},\theta)p(\mathcal{T}|\theta)p(\theta|\beta) \tag{6.2}$$

7 Program evaluation

In this thesis, each evolved hierarchy will itself represent a distribution. As the way to achieve this goal, the current implementation finds a program able to generate some data $\hat{\mathcal{X}}$ as similar to a given data set $\mathcal{X}$ as possible, minimizing

$\pi(\mathcal{X}|\hat{\mathcal{X}})$ as the way to maximize the similarity. We address two relevant questions in this approach:

1. The way π is defined: which statistics are used and the way they have been computed.
2. The way the inputs of the program are initialized: in principle, the $(N = 3)$ inputs of θ are generated from a uniform distribution.

7.1 Minimizing the distance between the summary statistics

Regarding the first of the previous two aspects, the most direct and easiest way to evaluate the quality of the solution is by generating random samples from the distribution represented by the program and comparing them with the original data that we want to approximate. However, there is not a unique way to design this evaluation, in fact, several methods have been devised and in what follows we discuss some of them.

Moments are widely used in statistics to characterize a distribution. They can ve computed from the data. In our approach, we used the first four moments. They were computed both, from the original data, and the sampled data. The moments computed were: the mean $(\mu_1, \hat{\mu_1})$, the variance $(\mu_2, \hat{\mu_2})$, but also known as $(\sigma^2, \hat{\sigma^2})$, the skewness $(\mu_3, \hat{\mu_3})$ and kurtosis $(\mu_4, \hat{\mu_4})$.

7.1.1 Direct evaluation through moments

The method computes the sum of the squared difference of the moments:

$$\sum_{n=1}^{4}(\mu_n - \hat{\mu_n})^2 \tag{7.1}$$

In this way, the lower the value of this sum, the more likely will be for the program to generate a distribution of the same properties that the observed data, at least in terms of the statistics considered.

Let us consider the next observed values with its corresponding moments

$$\mathbf{y} = [3, 9, 4, 5, 8, 2, 8, 1, 5, 1, 1, 5, 5, 8, 0, 1, 4, 9, 1, 1, 0, 3, 1, 3, 5, 5, 5, 8]$$

$$\mu_1 = 4.0667;\ \mu_2 = 7.9956;\ \mu_3 = 0.2949;\ \mu_4 = -1.1353$$

and the next $Poisson(\lambda = 2)$ distribution sample

$$\hat{\mathbf{y}} = [1, 0, 5, 0, 1, 1, 5, 2, 1, 2, 5, 2, 1, 2, 2, 2, 2, 0, 2, 2]$$

$$\mu_1 = 1.9;\ \mu_2 = 2.19;\ \mu_3 = 1.0053;\ \mu_4 = 0.3301$$

Finally, the error value for this example is

$$\sum_{n=1}^{4}(\mu_n - \hat{\mu_n})^2 = 41.0516$$

7.1.2 Normalized evaluation through moments

One of the problems that a direct evaluation such as the one previously described can face is the numerical explosion that can occur when the population is not able to adapt. It has been seen experimentally that when the mean of the observed data is very large, the evolved programs do not get close to that value, resulting in a fitness or total sum of the order of 10^4 or even higher. For example, consider an observed mean close to 300 and an individual represented by a normal distribution with parameters $\mu = 0$ and $\sigma = 1$, the expected *fitness* of that individual can exceed $9 \cdot 10^4$. This is, the evolution will get trapped in a poor solution or will take many generations to converge to an acceptable solution.

To overcome this obstacle, an option is to normalize the observed data, so that evolution is limited to using inputs from a bounded interval and thus, the search for the best individual could be focused on programs that produce distributions closer to the (normalized) observed data.

Standardization Using this approach, the sample mean is removed by scaling it until a unit variance is achieved.

It is computed as:

$$z = \frac{\mathbf{x} - \mu}{\sigma}$$

where μ and σ are the sample mean and standard deviation, respectively. With this transformation, the resulting values become independent of the unit, also having both the same dispersion and the same mean.

Following with the previous example, the new $\mathbf{y}'$ will be computed as

$$\mathbf{y}' = \frac{\mathbf{y} - 4.0667}{\sqrt{7.9956}} = [-0.38, -0.38, 1.74, -0.02, 0.33, 1.39, -0.73, 1.39, ...]$$

$$\mu_1 = 2.78 \cdot 10^{-8};\ \mu_2 = 1;\ \mu_3 = 0.2949;\ \mu_4 = -1.1353$$

and the $\hat{\mathbf{y}}$ is also standardized

$$\widehat{\mathbf{y}'} = \frac{\hat{\mathbf{y}} - 1.9}{\sqrt{2.19}} = [-0.61, -1.28, 2.09, -1.28, -0.61, -0.61, 2.09, 0.07, ...]$$

$$\mu_1 = 3.12 \cdot 10^{-8};\ \mu_2 = 1;\ \mu_3 = 1.0053;\ \mu_4 = 0.3301$$

The error value for the standardized example is

$$\sum_{n=1}^{4} (\mu_n - \widehat{\mu_n})^2 = 2.652$$

Normalization The sample values are transformed to a specific range, in this case, $[0, 1]$. This is done by applying the following

$$\mathbf{x}' = \frac{\mathbf{x} - \min(\mathbf{x})}{\max(\mathbf{x}) - \min(\mathbf{x})}$$

In a similar way but with normalization transformation,

$$\mathbf{y}' = \frac{\mathbf{y} - 0}{9 - 0} = [0.33, 0.33, 1., 0.44, 0.56, 0.89, 0.22, 0.89, ...]$$

$$\mu_1 = 1.9;\ \mu_2 = 2.19;\ \mu_3 = 0.2949;\ \mu_4 = -1.1353$$

and the $\hat{\mathbf{y}}$ is also normalized

$$\widehat{\mathbf{y}'} = \frac{\hat{\mathbf{y}} - 0}{5 - 0} = [0.2, 0., 1., 0., 0.2, 0.2, 1., 0.4, ...]$$

$$\mu_1 = 0.38;\ \mu_2 = 0.0876;\ \mu_3 = 1.0053;\ \mu_4 = 0.3301$$

Finally, after applying the normalization, the Equation 7.1.1 is computed to obtain the error

$$\sum_{n=1}^{4} (\mu_n - \widehat{\mu_n})^2 = 9.3825$$

7.1.3 Structural Similarity Index measure

So far, all the work developed in the thesis has been related to one-dimensional input data, i.e., distributions defined on vectors. However, an attempt was also made to apply this genetic procedure to two-dimensional data. Specifically, an attempt was made to obtain the hierarchical model underlying the digit "1" from the Modified National Institute of Standards and Technology database (MNIST) images [33]. To do so, another way of measuring the quality of a program had to be considered, since in this case, it was not enough to compute the statistics of the samples obtained because they did not provide information on the image structure: the sampled images had to have spatial coherence and these characteristics could not be obtained with metrics employed for the one-dimensional case.

Thus, the idea of using the Structural Similarity Index (SSI) [56] measure was born. This measure is very useful to quantify the visibility of the errors between a distorted image and a reference image. It is a technique widely used in television since it takes into account the degradation of images as perceptual phenomena, including those as illumination and contrast. Figure 13 shows two small perceptual modifications of the left-most image. However, both of them with the same Mean Squared Error (MSE) but different SSI.

However, this approach of sampling two-dimensional data was discarded because the results of the evolution were poor and no valid conclusion was reached.

Figure 13: Comparison of 8×8 MNIST "1" images with a small amount of distortions applied, all with MSE = 0.0002. The image on the left represents the original image. A random noise has been applied to the one in the center one (SSI = 0.66) while a constant has been added to the one on the right (SSI = 0.91).

7.2 Generating inputs

As for the generation of program inputs, it is necessary to make rough assumptions, as well explained in the paper [44]. The reason for these assumptions is that our learned program must work for any argument and not just a subset. Assuming that such a program works well for a few arguments, it is very likely to generalize to other configurations. Starting from Equation 6, one can account for this by choosing a finite small number N of parameters θ_n that yields the approximate objective:

$$\frac{1}{N}\sum_{n=1}^{N}\pi(\mathcal{X}_n|\hat{\mathcal{X}}_n,\theta_n)p(\hat{\mathcal{X}}_n|\mathcal{T},\theta_n)p(\mathcal{T}|\theta_n) \approx \int \pi(\mathcal{X}|\hat{\mathcal{X}},\theta_n)p(\hat{\mathcal{X}}|\mathcal{T},\theta)p(\mathcal{T}|\theta)p(\theta)d\theta \tag{7.2}$$

Notice that we assume that the parameters $\theta_1, ..., \theta_n$ have been generated from a given β and therefore we do not include β in Equation 7.2. Also, the parameter θ_n is introduced within the distance term to emphasize that the distance, i.e., the quality of the program will also depend on the choice of θ.

Therefore, by evaluating the statistics using different parameters we are able to identify robust programs, able to generalize. However, it is important to take into account that the input parameters could be generated in a different way. For example, one can pass many times the same initial parameters $(\theta_1, \theta_2, \theta_3)$. Since these programs are stochastic generators, even if the same inputs are passed the programs will generate different outputs. However, these vector is actually sampled from a $\mathcal{N}(0, 1)$ distribution in the current implementation. Each time the experiment is run, new but constant inputs are generated throughout the experiment.

8 Improving the quality of the programs

Another question also arises once we have obtained the best program capable of fitting the observed data, and that is, how to refine the accuracy of the model to represent the underlying target distribution of the data. We distinguish two ways to address this question:

1. Find the best set of inputs θ for the probabilistic program.
2. Find *a posteriori* distribution of θ given $\mathcal{X}$.

8.1 Optimizing the inputs

Once an individual has been evolved, such as the best program obtained using the inputs generated by a Normal distribution, one could try to find the set of input parameters $(\theta_1, \theta_2, \theta_3)$ such that it maximizes the fitness function.

With this approach, the program is fixed and a numerical optimization in the space of possible values for $(\theta_1, \theta_2, \theta_3)$ is performed. To evaluate each possible combination of $(\theta_1, \theta_2, \theta_3)$, one can sample the same program with the three values many times and use this as input for the fitness function. In order to optimize the inputs, Bayesian Optimization (BO) [51] could be used. This technique allows the evaluation of black-box functions that are costly to evaluate. BoTorch [5] is a library built on top of PyTorch for this purpose and should not be difficult to use with exactly the same fitness function.

The output of this approach would be an assignment for $(\theta_1, \theta_2, \theta_3)$ that is more likely to generate data $\hat{\mathcal{X}}$ very similar to $\mathcal{X}$.

8.2 Finding the *a posteriori* distribution

In this approach, we do not search for a fixed set of parameters θ. Instead, we would like to know which is a distribution of θ that produces samples similar to $\mathcal{X}$. We have assumed that this distribution depends on a latent variable β. Then, since we have the observed data $\mathcal{X}$, we would like to find the value of β that generates inputs θ such that when passed to our program, it most likely generates data similar to $\mathcal{X}$. This is an inference problem that can be addressed with different techniques.

One possible solution is to assume that the *a posteriori* distribution has some form, e.g., $\mathcal{N}(\mu, \sigma^2)$, and find the parameters μ and σ^2. If we have the *a posteriori* distribution, then we can generate inputs that when passed to our program will likely generate data similar to $\mathcal{X}$.

The difficulty is that the procedures for finding this *a posteriori* distributions are complex, and they do not always converge. Pyro uses *guide* functions as a way to propose a family of distributions to which the *a posteriori* distribution is expected to belong. However, in many real world problems it is not possible to know in advance which is the family of distributions.

One way to evaluate the quality of the evolved programs would be, to generate a number of vectors $(\theta_1, \theta_2, \theta_3)$ from the *a posteriori* distribution, pass these parameters to the evolved program and compute the fitness in the usual way,

with the samples generated by the programs. Ideally, since the distribution was learned using $\mathcal{X}$ as the observations, we would expect the fitness to be better than if the input parameters $(\theta_1, \theta_2, \theta_3)$ were selected using an arbitrary distribution.

It is worth commenting that this optimization variant has been implemented, while the numerical optimization of the input parameters has been left as future work.

9 Experiments

9.1 Motivation

When facing the experimental part, it becomes necessary to raise the questions to be answered with these experiments:

- Is it possible to learn a model that approximates the observed data?
- Is it possible to learn a model as good as the already known model?
- Is there a significant improvement in the evolution with the passing of generations?

To answer these questions, the workflow shown in Figure 14 has been followed for data ingestion, learning and evaluation.

The methodology that has been followed in all experiments is described in the next paragraphs.

Preprocessing Despite the variety of the experiments, the observed datasets are loaded individually in a similar way, ending with a one-dimensional tensor representing the observed values. Then, holdout is applied at 65% to split the data at random into train and test instances. The proposed benchmarks run the same experiments three times: a first one without scaling, one normalizing and the last one, standardizing the data. Finally, the test instance will be used together with the values sampled by the evolved programs, to calculate the fitness of each program, but in no case will this test set be modified or processed.

Evolution The evolution process has been initialized with the following parameters. The number of individuals in the initial population is 50, evolving during 250 generations. The crossover and mutation probabilities are set to 0.5 and 0.1 respectively, while the tournament size is 3. Also, some constraints have been applied to the genetic programs. The minimum and maximum depth of the three have been set to 1 and 10 respectively.

9.2 Use cases

As an initial baseline, it will be interesting to probe if our system is able to learn probabilistic programs using data sampled from simple distributions. Also, several real-world datasets have been considered to test our assumptions. All these scenarios have been taken from the work in [52]. Our goal when using these datasets is to model observed data taken from realistic scenarios.

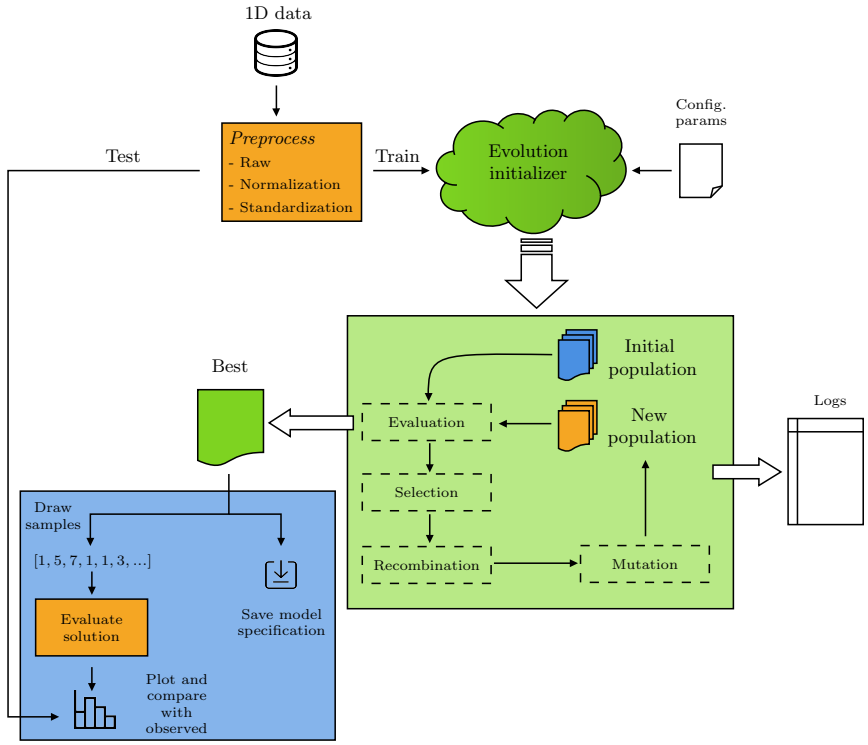
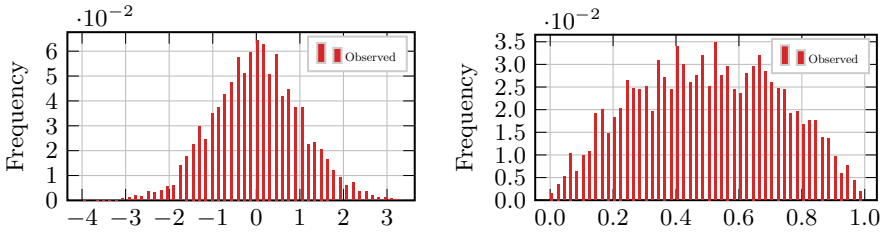


Figure 14: Experimentation workflow used to evolve probabilistic programs that represent a distribution.

9.2.1 Case #1: Learning simple distributions

The proposed distributions to learn are the $\mathcal{N}(0, 1)$ and $Beta(2, 2)$ distributions, sampling 2500 data points at random, as shown in Figures 15a and 15b.



(a) $\mathcal{N}(0, 1)$ distribution samples.

(b) $Beta(2, 2)$ distribution samples.

Figure 15: Distribution samples for Normal and Beta distributions.

9.2.2 Case #2: *Average Minimum Temperature in Scotland*

This dataset provides the average minimum temperature in Scotland in month November for the years 1884 - 2020 and it was retrieved by Met Office National Climate Information Centre[4]. Figure 16 shows the histogram of data.

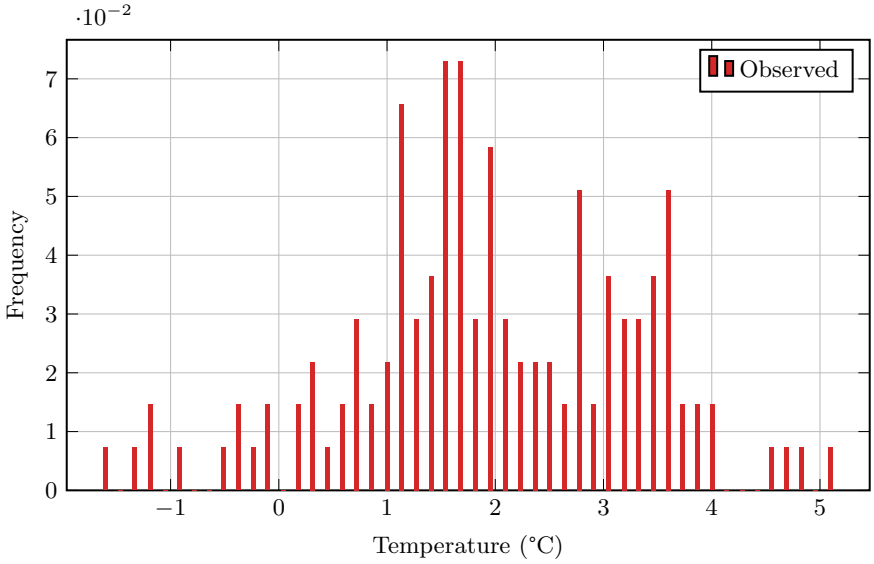


Figure 16: Minimum November temperatures for the 1884 - 2020 period in Scotland.

9.2.3 Case #3: *Modelling the precipitation*

Correctly modeling and understanding meteorological processes can help mitigate the effects of climate change and promote the correct use of the planet's natural resources. One of them, vital for human survival, is water, and in many places, having it is a privilege. Much has been studied about the phenomenon of precipitation, both from the physical and meteorological point of view and from the statistical point of view [57], being applied to hydrological, agricultural or ecosystem modeling. Stochastic models are very common in the statistical approach, known as "weather generators" [57], since they can generate synthetic data series from observed values.

As a third use case and a more local example, we try to model the precipitation from Biscay, precisely, in Punta Galea. There are exhaustive works that have already attempted to model precipitation in the Basque Country, such as the study [40] in which the authors fit a characteristic polynomial for the Cantabrian

[4] https://www.metoffice.gov.uk/pub/data/weather/uk/climate/datasets/Tmin/date/Scotland.txt

Basin and the Ebro Basin, taking the daily precipitation from the years 1981 to 1988, or the much more detailed work [38], using pluviometric data from the years 1961 to 2000 to analyze and reconstruct the series using the reanalysis[5] data of the ERA-40 [55], of the European Centre for Medium-Range Weather Forecasts (ECMWF), and thus compare with the data obtained from different regional models.

Here, however, in order not to extend the scope of the project, we limit ourselves exclusively to fit a distribution to the Punta Galea precipitation data, a cape located near the cities of Sopela and Getxo, distinguished by its cliffs and famous among surfers. All the data has been scraped and downloaded from the Basque Meteorological Agency[6] and Open Data Euskadi, a government transparency platform.

The data downloaded contains many variables, such as precipitation, temperature, humidity, irradiance and atmospheric pressure, all of them recorded from a meteorological station placed in Punta Galea every 5 minutes. Only the accumulated rainfall values over a period of 24 hours instead, from January 2019 to July 2021 has been taken into account. Figure 17 shows the histogram for the data.

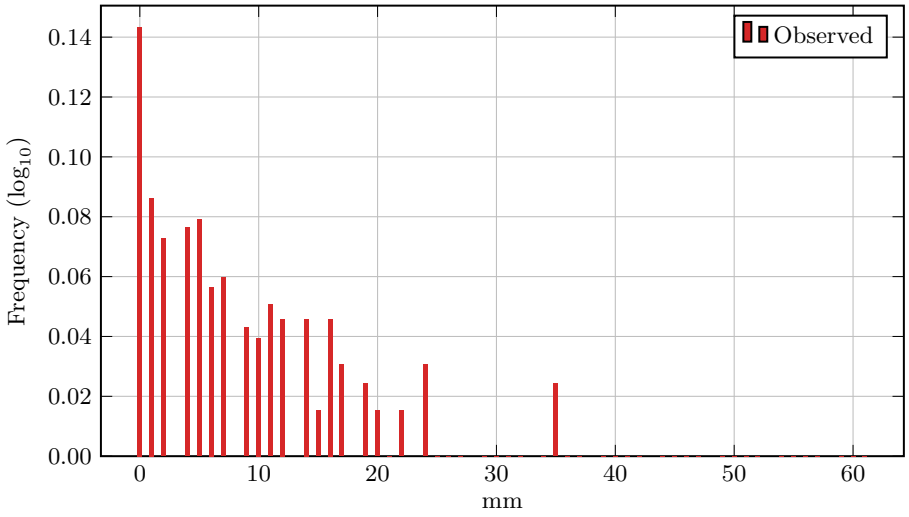


Figure 17: Cumulated rainfall during 24 hours in Punta Galea (Biscay).

[5] In meteorology, reanalysis combines data from numerical forecast models with observations from around the world into a globally complete and coherent data set using the laws of physics. This principle, called data assimilation, combines an earlier forecast with newly available observations in an optimal way to produce a new optimal estimate of the state of the atmosphere, called an analysis, from which an updated and improved forecast is issued. Reanalysis works in a similar way, but with a reduced resolution that allows for a data set going back several decades.

[6] https://www.euskalmet.euskadi.eus/observacion/datos-de-estaciones

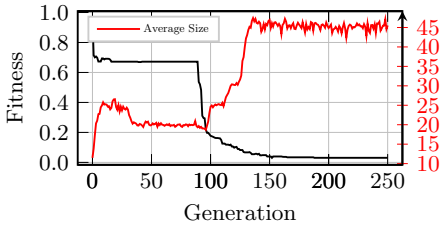
10 Results

In this section the obtained results are shown, both graphical comparisons and the best individual evolved with our method. The code used to launch these experiments is available in Appendix B.

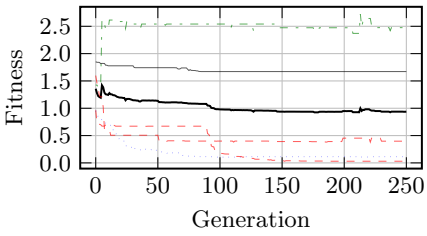
10.1 Learning simple distributions

The first distribution learned in the experiments performed was the Normal distribution. As already mentioned, the evolution has taken into account 250 generations in which the models have been evolving and improving to fit the data in an optimal way. It is necessary to emphasize that for this case, it has not been allowed to learn using the normal distribution, since it would be "facilitating" the search for the best program and would not demonstrate the capability of the system. That is, excluding the original distribution of the data forces the genetic search to actually search for a combination that generates the closest match to the target distribution, without resorting to the easy solution of using precisely that distribution as a component of the tree.

This process can be visualized in Figure 18a. This graph shows the fitness of the run that obtained the best model together with the average size of the evolved programs. On the other hand, Figure 18b shows the fitness of the 5 runs of the experiment, where the dotted lines represent each of the repetitions.



(a) Logbook of the best model.



(b) Evolution logbook after 5 repetitions. Each dashed line corresponds to a single run of the evolution. The black solid one, refers to the mean.

Figure 18: Logbooks for the Normal distribution learning.

The best evolved model is shown in Figure 19. It is, after all, a program that models a Beta distribution. However, the reader may notice that the values sampled by this model are not part of the distribution domain, as Figure 20a points out. Recall that in each experiment, the data are manipulated in three different ways: one by using the raw data; another by normalizing to an interval $(0, 1)$; and finally, by standardizing the data. Thus, when the data were normalized, evolution was able to choose a Beta distribution within the bounded domain of normalized values that significantly improved fitness. Then, it is necessary to

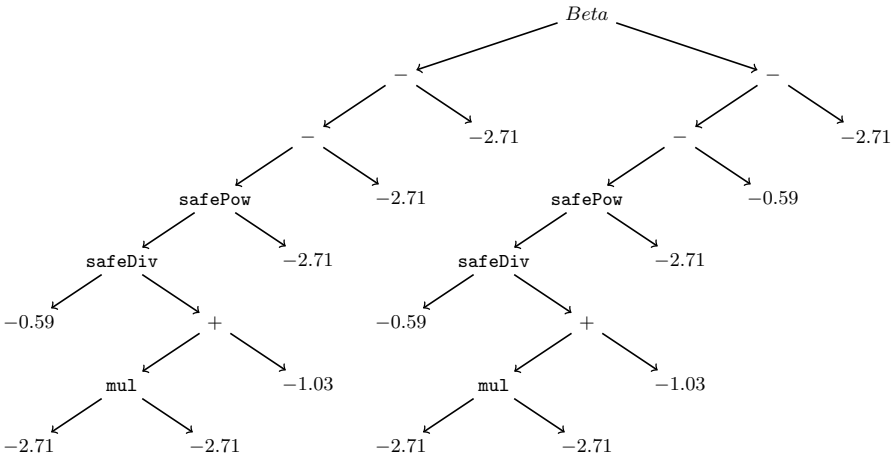
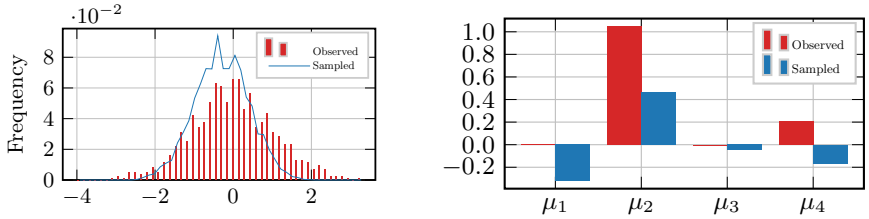


Figure 19: Best evolved model for the Normal distribution learning. Intermediate type conversion steps have been omitted.

rescale to the original domain of the observed values the values sampled by the distribution, and this is exactly what Figure 20a shows: the Beta sampled values rescaled to the original domain of the observed data.



(a) Comparison between the true $\mathcal{N}(0, 1)$ sampled values (red) and the best model sampled values (blue).

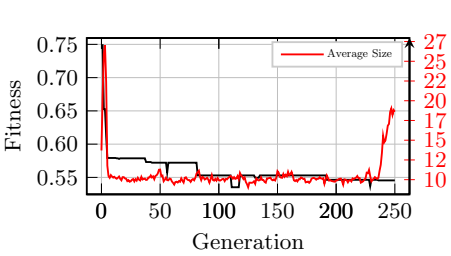
(b) Evolved model's summary statistics vs. test set.

Figure 20: Sampled values, observed values and summary statistics for the Normal distribution learning.

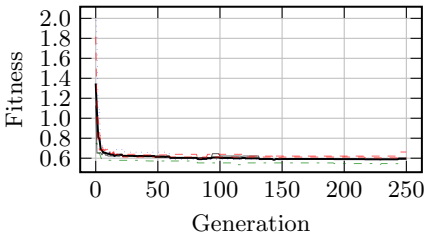
Figure 20b provides information about the statistics of the observed and sampled values for the best model for comparison. It is a good approximation even though there is some difference between the two.

With respect to learning the Beta distribution, as in the previous case, the Beta distribution has not been allowed in this training. In this case, similar performance has been achieved, standardizing the values to then learn a Normal distribution seems to be very efficient and very fast, as seen in Figure 21a, where all runs fit almost immediately in the first iterations of the evolution. The learned

model, Figure 22, turns out to be a simple Normal distribution whose sampled values resemble the reference values, as drawn in Figure 23a. This is attested by the statistics in Figure 23b, whose mean (μ_1) and variance (μ_2) are practically identical.



(a) Logbook of the best model.



(b) Evolution logbook after 5 repetitions. Each dashed line corresponds to a single run of the evolution. The black solid one, refers to the mean.

Figure 21: Logbooks for the Beta distribution learning.

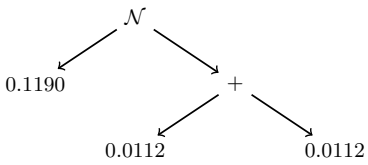
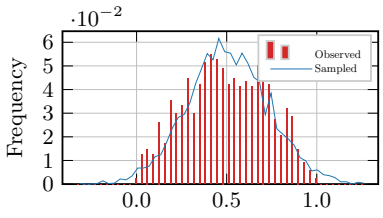
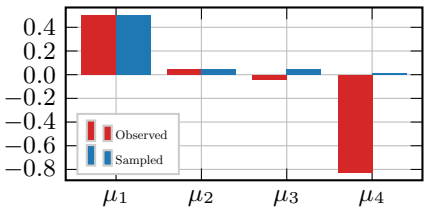


Figure 22: Best evolved model for the Beta distribution learning. Intermediate type conversion steps have been omitted.



(a) Comparison between the true $Beta(2, 2)$ sampled values (red) and the best model sampled values (blue).



(b) Evolved model's summary statistics vs. test set.

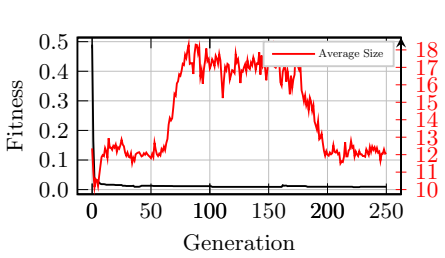
Figure 23: Sampled values, observed values and summary statistics for the Beta distribution learning.

It can be seen from these two simple experiments that the system proposed in this study is quite capable of learning models that mimic arbitrary values, in this case, the known starting distribution.

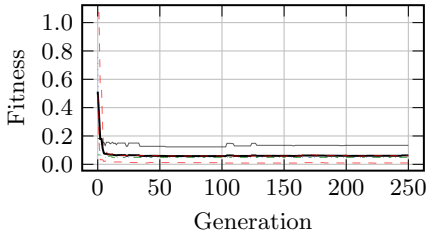
10.2 *Average Minimum Temperature in Scotland*

For the average temperature modeling experiment, we had much less data than in the previous experiments. However, that did not pose much difficulty to the system, as Figure 24a shows. In very few iterations it has managed to practically converge and this is confirmed by the different repetitions of the experiment that have been carried out, shown in Figure 24b.

In this particular case, the evolved program has quickly learned from the input data and managed to mimic it quite closely, as shown by the statistics in Figure 26b. After obtaining the best program, it has been sampled and compared with the original data. It is necessary to comment that the best program has been achieved by using standardization prior to data ingestion.



(a) Logbook of the best model.



(b) Evolution logbook after 5 repetitions. Each dashed line corresponds to a single run of the evolution. The black solid one, refers to the mean.

Figure 24: Logbooks for the *temperature* problem.

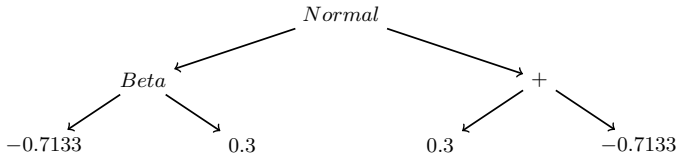
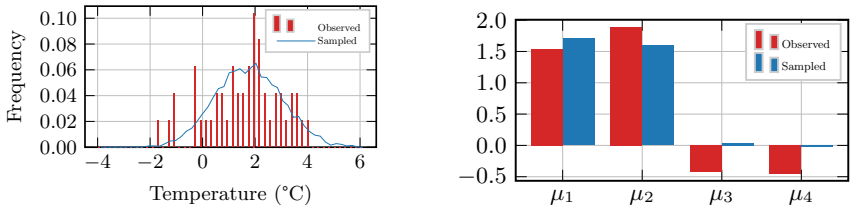


Figure 25: Best evolved model for the *temperature* problem. Intermediate type conversion steps have been omitted.

Again, a fairly good and accurate result is achieved without getting too complex a program, as shown in Figure 25. This is a Normal distribution, whose parameters *mu* and *sigma* are parameterized according to a Beta distribution



(a) Comparison between the observed values (red) and the best model sampled values (blue).

(b) Evolved model's summary statistics vs. test set.

Figure 26: Sampled values, observed values and summary statistics for the *temperature* problem.

and a scalar. Strictly speaking, the program shown in Figure 25 is not quite correct, since a Beta distribution does not admit negative values among its arguments, as does *sigma* in the Normal distribution. However, these steps are omitted in the figure. Strictly, the model would be defined as Equation 10.2.

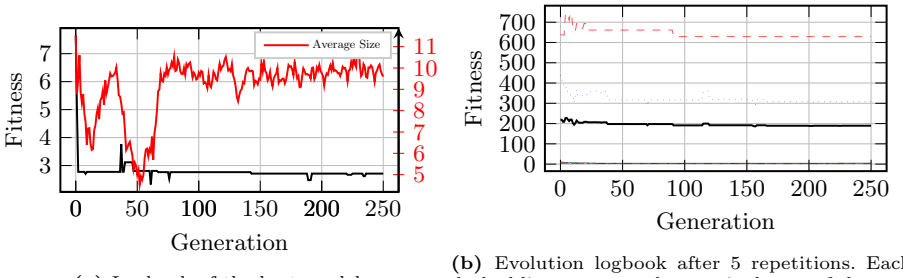
$$\begin{aligned} Y &\sim \mathcal{N}(\mu, \sigma) \\ \mu &\sim Beta(a, b) \\ \sigma &= max(10^{-7}, |0.3 - 0.7133|) \\ a &= max(10^{-7}, | - 0.7133|) \\ b &= 0.3 \end{aligned} \tag{10.1}$$

10.3 *Modelling the precipitation*

As the last problem analyzed, we have the modeling of precipitation. In contrast to the previous experiments, this evolution has not achieved as good results as one might expect, although it is quite close. The experiment that has achieved the best program, as shown in Figure 27a, has converged quickly in the first few iterations. In contrast, it can be seen that not all runs have done equally well and in fact, some have produced very poor results as illustrated in Figure 27b.

It is important to understand well the graph in Figure 17 in order not to mislead. First, it should be noted that the data has been normalized, so that the frequency of all values adds up to 1 and second, a logarithmic transformation has been applied to this normalization for one reason: $\approx 73\%$ of the values are 0 (no precipitation has been recorded) and to better visualize the rest of the frequencies, this transformation has been chosen. However, in the comparison between the test set and the values sampled by the best program in Figure 29a, this logarithmic transformation has not been applied. That is why both figures have different representation.

Moreover, this is the reason why evolution has opted for a χ^2 distribution, model drawn in Figure 28. When the only parameter of this distribution takes small (natural) values, for example 1, the slope of the probability density function



(a) Logbook of the best model.

(b) Evolution logbook after 5 repetitions. Each dashed line corresponds to a single run of the evolution. The black solid one, refers to the mean.

Figure 27: Logbooks for the *precipitation* problem.

turns out to be very steep, being very suitable for this problem. Note that in Figure 28 the type conversion has been omitted, and mathematically, the model would be equal to $Y \sim \chi^2(1)$.

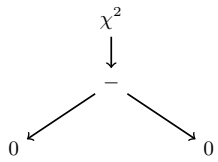
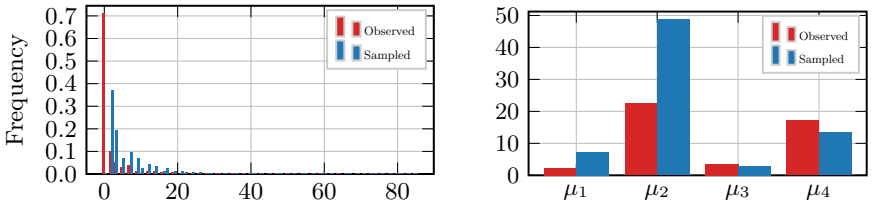


Figure 28: Best evolved model for the *precipitation* problem. Intermediate type conversion steps have been omitted.



(a) Comparison between the observed values (red) and the best model sampled values (blue).

(b) Evolved model's summary statistics vs. test set.

Figure 29: Sampled values, observed values and summary statistics for the *precipitation* problem.

Finally, Figure 29b shows the quality of the best individual, for which, as already mentioned, there are considerable differences between the evolved model and the observed data.

Part VI

Conclusions

11 Summary

This thesis has addressed the problem of learning probabilistic programs from a given set of observations. In order to do so, we have designed and implemented methods to carry out such learning using EAs.

In the first part of the thesis, an introduction to the key concepts of the project has been given, along with an explanation of probabilistic programming accompanied by some examples and a brief review of the GA and one of its variants, GP. Then, related research work in these two fields and practical application cases (such as TPOT) have been discussed.

Regarding the development of the project and the implementation, a grammar accompanied by types and functions has been proposed for the construction of syntactically valid programs within the evolutionary environment, as well as a formal definition of the problem to be addressed, a discussion about the evaluation of the programs and possible options to optimize such programs. In the last part of the thesis, we have evaluated the GP approach on simulated and real data, analyzing the performance of the evolution and inspecting the best solutions.

12 Conclusions

Developing models that fit observed data is usually not an easy task. Many standards have been established over the years by the experience of experts, such as the use of Weibull and Rayleigh functions to describe the frequency of wind speed distributions [10], normal distributions following rainfall and river discharges of long duration (e.g. monthly or annual) [47] or even the use of the Poisson distribution to model the goals in a soccer match [23]. However, it may be the case that these models do not perfectly capture all the casuistry or that new models are better. The study carried out in this thesis aims to take a step in that direction, trying to discover new models that fit observed data without the need for any prior knowledge of the data, since the proposed system is in charge of learning all these relationships.

One of the main objectives of this work was to verify that the theoretical assumptions about learning probabilistic programs by means of evolutionary algorithms were feasible and close to what was observed or expected. This has been

demonstrated in the three experiments presented. Firstly, probabilistic distributions or programs capable of resembling the observed data is obtained, which are also learned in a relatively short interval of time, all this without the need to know beforehand the structure or nature of the data itself. Second, the grammar developed does not consider all the existing distributions nor all the operations that may occur between distributions or values, so it is a simple first approximation of the potential of the work. With a small number of distributions, such as that of the grammar presented, a modest number of common problems can be modeled, but moreover, the inclusion of new distributions and operations in the proposed system would not be difficult, allowing an even greater search space, enriching the models at the same time that they can be made more complex.

13 Future work

As future work, we identify a number of research directions:

1. The addition of more variables or dimensions in the learning of the problems can be considered. So far, only one variable has been taken into account in each experiment and including more information in the learning process could be useful, although increasing the complexity.
2. It may also be of interest to model the *a posteriori* distribution with which we sample the parameters at the beginning of each experiment. In the current implementation, we use $\mathcal{N}(0, 1)$ to sample the inputs, but it may be interesting to know which parameters (μ, σ^2) are optimal to replace such a Gaussian by $\mathcal{N}(\mu, \sigma^2)$.
3. A multi-objective evaluation of the programs can also be considered. In this study, only the evaluation using the statistics of the generated samples has been considered, but in turn, a second objective could be added to minimize the complexity of the trees, such as height.
4. This system could also be used to learn problems of a higher dimensionality, as we have tried with the example of the MNIST images, or even contemplate a problem that has some temporal component. For example, within weather forecasting, numerical models provide information both spatially and temporally, and even at atmospheric levels such as heights and pressures. In other words, 2D, 3D and 4D data that could be interesting to model with distributions.
5. Studying other crossover and mutation operators could speed up the learning process by analyzing the search space they can offer, and provide higher quality solutions.
6. Another method can be implemented to optimize the input parameters of the distribution, using BO, as explained in Section 8.1.

Bibliography

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] M. Affenzeller. Segregative Genetic Algorithms (SEGA): A hybrid superstructure upwards compatible to genetic algorithms for retarding premature convergence. *Int. J. Comput. Syst. Signal*, 2(1):16–30, 2001.

[3] M. Affenzeller and S. Wagner. SASEGASA: A new generic parallel evolutionary algorithm for achieving highest quality results. *Journal of Heuristics*, 10(3):243–267, 2004.

[4] A. Agapitos, R. Loughran, M. Nicolau, S. Lucas, M. O'Neill, and A. Brabazon. A survey of statistical machine learning elements in genetic programming. *IEEE Transactions on Evolutionary Computation*, 23(6):1029–1048, 2019.

[5] M. Balandat, B. Karrer, D. R. Jiang, S. Daulton, B. Letham, A. G. Wilson, and E. Bakshy. BoTorch: A Framework for Efficient Monte-Carlo Bayesian Optimization. In *Advances in Neural Information Processing Systems 33*, 2020.

[6] M. Betancourt. A conceptual introduction to Hamiltonian Monte Carlo. *arXiv preprint arXiv:1701.02434*, 2017.

[7] E. Bingham, J. P. Chen, M. Jankowiak, F. Obermeyer, N. Pradhan, T. Karaletsos, R. Singh, P. A. Szerlip, P. Horsfall, and N. D. Goodman. Pyro: Deep Universal Probabilistic Programming. *J. Mach. Learn. Res.*, 20:28:1–28:6, 2019.

[8] F. Briggs and M. O'neill. Functional genetic programming with combinators. In *Proceedings of the Third Asian-Pacific workshop on Genetic Programming, ASPGP*, pages 110–127, 2006.

[9] S. Chib and E. Greenberg. Understanding the Metropolis-Hastings algorithm. *The American Statistician*, 49(4):327–335, 1995.

[10] K. Conradsen, L. Nielsen, and L. Prahm. Review of Weibull statistics for estimation of wind speed distributions. *Journal of Applied Meteorology and Climatology*, 23(8):1173–1183, 1984.

[11] L. De Raedt and A. Kimmig. Probabilistic programming concepts. *arXiv preprint arXiv:1312.4328*, 2013.

[12] B. P. Evans, B. Xue, and M. Zhang. What's inside the black-box? A genetic programming method for interpreting complex machine learning models. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1012–1020, 2019.

[13] C. Ferri-Ramírez, J. Hernández-Orallo, and M. J. Ramírez-Quintana. Incremental learning of functional logic programs. In *International Symposium on Functional and Logic Programming*, pages 233–247. Springer, 2001.

[14] P. Flener and S. Yılmaz. Inductive synthesis of recursive logic programs: Achievements and prospects. *The Journal of Logic Programming*, 41(2-3):141–195, 1999.

[15] A. E. Gelfand. Gibbs sampling. *Journal of the American Statistical Association*, 95(452):1300–1304, 2000.

[16] S. Geman and D. Geman. Stochastic relaxation, Gibbs distributions, and the Bayesian restoration of images. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, (6):721–741, 1984.

[17] D. E. Goldberg and J. H. Holland. Genetic Algorithms and Machine Learning. 1988.

[18] D. E. Goldberg, R. Lingle, et al. Alleles, loci, and the traveling salesman problem. In *Proceedings of an International Conference on Genetic Algorithms and their Applications*, volume 154, pages 154–159. Lawrence Erlbaum Hillsdale, NJ, 1985.

[19] N. Goodman, V. Mansinghka, D. M. Roy, K. Bonawitz, and J. B. Tenenbaum. Church: a language for generative models. *arXiv preprint arXiv:1206.3255*, 2012.

[20] N. D. Goodman and A. Stuhlmüller. The Design and Implementation of Probabilistic Programming Languages. http://dippl.org, 2014. Accessed: 2021-4-17.

[21] A. D. Gordon, T. A. Henzinger, A. V. Nori, and S. K. Rajamani. Probabilistic programming. In *Future of Software Engineering Proceedings*, pages 167–181. 2014.

[22] R. Grosse, R. R. Salakhutdinov, W. T. Freeman, and J. B. Tenenbaum. Exploiting compositionality to explore a large space of model structures. *arXiv preprint arXiv:1210.4856*, 2012.

[23] A. Heuer, C. Mueller, and O. Rubner. Soccer: Is scoring goals a predictable Poissonian process? *EPL (Europhysics Letters)*, 89(3):38007, 2010.

[24] J. H. Holland et al. *Adaptation in Natural and Artificial Aystems: an Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*. MIT press, 1992.

[25] K. Kersting. An inductive logic programming approach to statistical relational learning. *AI Communications*, 19(4):389–390, 2006.

[26] E. Kitzelmann, U. Schmid, R. Olsson, and L. P. Kaelbling. Inductive synthesis of functional programs: An explanation based generalization approach. *Journal of Machine Learning Research*, 7(2), 2006.

[27] D. Koller and N. Friedman. *Probabilistic Graphical Models: Principles and Techniques*. MIT press, 2009.

[28] J. R. Koza. Hierarchical genetic algorithms operating on populations of computer programs. In *IJCAI*, volume 89, pages 768–774, 1989.

[29] J. R. Koza and J. R. Koza. *Genetic Programming: on the Programming of Computers by Means of Natural Selection*, volume 1. MIT press, 1992.

[30] K. Krawiec. Genetic programming-based construction of features for machine learning and knowledge discovery tasks. *Genetic Programming and Evolvable Machines*, 3(4):329–343, 2002.

[31] A. Kucukelbir, R. Ranganath, A. Gelman, and D. M. Blei. Automatic variational inference in Stan. *arXiv preprint arXiv:1506.03431*, 2015.

[32] W. B. Langdon. Size fair and homologous tree genetic programming crossovers. *Genetic programming and Evolvable Machines*, 1(1/2):95–119, 2000.

[33] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[34] P. Liang, M. I. Jordan, and D. Klein. Learning programs: A hierarchical Bayesian approach. In *Proceedings of the 7th International Conference on Machine Learning (ICML-10)*, pages 639–646, 2010.

[35] D. Lin, E. Dechter, K. Ellis, J. B. Tenenbaum, and S. H. Muggleton. Bias reformulation for one-shot function induction. *Frontiers in Artificial Intelligence and Application*, pages 525–530, 2014.

[36] V. Mansinghka, D. Selsam, and Y. Perov. Venture: a higher-order probabilistic programming platform with programmable inference. *arXiv preprint arXiv:1404.0099*, 2014.

[37] V. K. Mansinghka et al. *Natively Probabilistic Computation*. PhD thesis, Massachusetts Institute of Technology, Department of Brain and Cognitive Sciences, 2009.

[38] R. Moncho, G. Chust, V. Caselles Miralles, et al. Análisis de la precipitación del País Vasco en el período 1961-2000 mediante reconstrucción espacial. *Nimbus: Revista de climatología, meteorología y paisaje*, (23):149–170, 2009.

[39] S. Muggleton and L. De Raedt. Inductive logic programming: Theory and methods. *The Journal of Logic Programming*, 19:629–679, 1994.

[40] J. G. Muñiz, A. Auzmendi, and J. L. Siendones. Regionalización de la precipitación en el País Vasco: Aplicación del modelo de análisis regional de lluvias. *Geogaceta*, 10:128–130, 1991.

[41] R. S. Olson and J. H. Moore. TPOT: A tree-based pipeline optimization tool for automating machine learning. In F. Hutter, L. Kotthoff, and J. Vanschoren, editors, *Proceedings of the 2016 Workshop on Automatic Machine Learning, AutoML2016, co-located with 33rd International Conference on Machine Learning (ICML 2016), New York City, NY, USA, June 24, 2016*, volume 64 of *JMLR Workshop and Conference Proceedings*, pages 66–74. JMLR.org, 2016.

[42] R. Olsson. Inductive functional programming using incremental program transformation. *Artificial Intelligence*, 74(1):55–81, 1995.

[43] J. Pearl. *Probabilistic Reasoning in Intelligent Systems - Networks of Plausible Inference*. Morgan Kaufmann series in representation and reasoning. Morgan Kaufmann, 1989.

[44] Y. N. Perov and F. D. Wood. Learning probabilistic programs. *arXiv preprint arXiv:1407.2646*, 2014.

[45] R. Poli, W. B. Langdon, and N. F. McPhee. *A Field Guide to Genetic Programming*. lulu.com, 2008.
[46] A. Prékopa. Probabilistic programming. *Handbooks in Operations Research and Management Science*, 10:267–351, 2003.
[47] H. Ritzema. Drainage Principles and Applications, Publication 16. *International Institute for Land Reclamation and Improvement (ILRI), Wageningen, The Netherlands*, 3(39):1–47, 1994.
[48] D. Roth. On the hardness of approximate reasoning. *Artificial Intelligence*, 82(1-2):273–302, 1996.
[49] J. Salvatier, T. V. Wiecki, and C. Fonnesbeck. Probabilistic programming in Python using PyMC3. *PeerJ Computer Science*, 2:e55, 2016.
[50] U. Schmid and F. Wysotzki. Induction of recursive program schemes. In *European Conference on Machine Learning*, pages 214–225. Springer, 1998.
[51] J. Snoek, H. Larochelle, and R. P. Adams. Practical Bayesian optimization of machine learning algorithms. *Advances in Neural Information Processing Systems*, 25, 2012.
[52] E. Taka, S. Stein, and J. H. Williamson. Increasing Interpretability of Bayesian Probabilistic Programming Models Through Interactive Representations. *Frontiers Comput. Sci.*, 2:567344, 2020.
[53] A. Teller. Turing completeness in the language of genetic programming with indexed memory. In *Proceedings of the First IEEE Conference on Evolutionary Computation. IEEE World Congress on Computational Intelligence*, pages 136–141. IEEE, 1994.
[54] D. Tran, A. Kucukelbir, A. B. Dieng, M. Rudolph, D. Liang, and D. M. Blei. Edward: A library for probabilistic modeling, inference, and criticism. *arXiv preprint arXiv:1610.09787*, 2016.
[55] S. M. Uppala, P. Kållberg, A. J. Simmons, U. Andrae, V. D. C. Bechtold, M. Fiorino, J. Gibson, J. Haseler, A. Hernandez, G. Kelly, et al. The ERA-40 re-analysis. *Quarterly Journal of the Royal Meteorological Society: A journal of the Atmospheric Sciences, Applied Meteorology and Physical Oceanography*, 131(612):2961–3012, 2005.
[56] Z. Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli. Image quality assessment: from error visibility to structural similarity. *IEEE Transactions on Image Processing*, 13(4):600–612, 2004.
[57] D. S. Wilks and R. L. Wilby. The weather generation game: a review of stochastic weather models. *Progress in Physical Geography*, 23(3):329–357, 1999.
[58] S. Winkler, M. Affenzeller, and S. Wagner. Advanced genetic programming based machine learning. *Journal of Mathematical Modelling and Algorithms*, 6(3):455–480, 2007.
[59] F. Wood, J. W. Meent, and V. Mansinghka. A new approach to probabilistic programming inference. In *Artificial Intelligence and Statistics*, pages 1024–1032. PMLR, 2014.
[60] J. R. Woodward and R. Bai. Why evolution is not a good paradigm for program induction: a critique of genetic programming. In *Proceedings of the first ACM/SIGEVO Summit on Genetic and Evolutionary Computation*, pages 593–600. 2009.

Acronyms

AI	Artificial Intelligence.
AutoML	Automated Machine Learning.
BO	Bayesian Optimization.
DEAP	Distributed Evolutionary Algorithms in Python.
DL	Deep Learning.
EA	Evolutionary Algorithm.
ECMWF	European Centre for Medium-Range Weather Forecasts.
GA	Genetic Algorithm.
GP	Genetic Programming.
GPU	Graphics Processing Unit.
IP	Inductive Programming.
KL	Kullback-Leibler.
LP	Linear Programming.
MCMC	Markov Chain Monte Carlo.
ML	Machine Learning.
MNIST	Modified National Institute of Standards and Technology database.
MSE	Mean Squared Error.
PDF	Probability Density Function.
PPL	Probabilistic Programming Language.
SASEGASA	Self Adaptive Segregative Genetic Algorithm.
SEGA	Segregative Genetic Algorithm.
SSI	Structural Similarity Index.
TPOT	Tree-based Pipeline Optimization Tool.
XAI	Explainable AI.

Appendices

A Probabilistic Programming in other research areas

The first area where the term probabilistic programming has been used refers to the field of mathematical optimization, similar to the well known Linear Programming (LP), where a linear objective function needs to be maximized or minimized, which are subject to some linear constraints.

Linear programs are problems that can be represented in canonical form as shown in Equation A.1:

$$\begin{aligned} \text{maximize} \quad & \boldsymbol{c}^T \boldsymbol{x} \\ \text{subject to} \quad & A\boldsymbol{x} \leq \boldsymbol{b} \\ & \boldsymbol{x} \geq 0 \end{aligned} \tag{A.1}$$

where $\boldsymbol{c}$ and $\boldsymbol{b}$ are given vectors, A is a given matrix and $\boldsymbol{x}$, a vector whose components need to be determined, optimizing $\boldsymbol{c}^T\boldsymbol{x}$. Due to the $A\boldsymbol{x} \leq \boldsymbol{b}$ and $\boldsymbol{x} \geq 0$ inequalities, the constraints specify a convex polytope over which the objective function is to be optimized. Geometrically, these define the feasible region, this is, the set of all points that satisfy the constraints. However, an optimal solution may not exist due to different reasons: inconsistent constraint definitions, an unbounded polytope, etc.

The formulation however, for the mathematical optimization techniques known as Probabilistic Programming [46], replaces the classical constraints with probabilistic ones, as shown in Equation A.2:

$$\begin{aligned} \text{maximize} \quad & \boldsymbol{c}^T \boldsymbol{x} \\ \text{subject to} \quad & P(T\boldsymbol{x} \geq \boldsymbol{\xi}) \geq \boldsymbol{p} \\ & A\boldsymbol{x} \leq \boldsymbol{b} \\ & \boldsymbol{x} \geq 0 \end{aligned} \tag{A.2}$$

where $\boldsymbol{\xi}$ is a random vector, $\boldsymbol{p}$ are given numbers and T a given matrix.

As pointed by [46], it is possible to find an application of these techniques in a flood control reservoir system design problem. In the simplest version, two reservoir sites where capacities x_1 and x_2 have to be determined in order to protect a downstream area from flood. If ξ_1 and ξ_2 are the water amount to be retained by the reservoirs, then the flood will be retained if and only if $x_1 + x_2 \geq \xi_1 + \xi_2, x_2 \geq \xi_2$ are satisfied, but since ξ_1 and ξ_2 are random variables, the fulfilment of these inequalities can be guaranteed only on a probability level p, chosen by ourselves. This can be modeled as Equation A.3.

$$\begin{aligned} \text{minimize} \quad & c(x_1, x_2) \\ \text{subject to} \quad & P\begin{pmatrix} x_1 + x_2 \geq \xi_1 + \xi_2 \\ x_2 \geq \xi_2 \end{pmatrix} \geq p \\ & 0 \leq x_1 \leq V_1 \\ & 0 \leq x_2 \leq V_2 \end{aligned} \tag{A.3}$$

where V_1 and V_2 are upper bounds determined by the local geographic situation.

B Experiment replication

The following appendix shows the arguments used to launch each of the experiments described in the paper. All the code is openly available in the following repository: https://github.com/r3v1/ec-ppl.

B.1 Case #1: Learning simple distributions

```
# Normal distribution
python src/benchmark.py --bench normal --repeat 5 --
    generations 250 --id svi --loss simple --optimizer svi

# Beta distribution
python src/benchmark.py --bench beta --repeat 5 --generations
     250 --id svi --loss simple --optimizer svi
```

B.2 Case #2: *Average Minimum Temperature in Science*

```
# Temperature
python src/benchmark.py --bench temperature --repeat 5 --
    generations 250 --id svi --loss simple --optimizer svi
```

B.3 Case #3: *Modelling the precipitation*

```
# Precipitation
python src/benchmark.py --bench precipitation --repeat 5 --
    generations 250 --id svi --loss simple --optimizer svi
```