NOVA IMS
Information Management School

DOCTORAL PROGRAMME

Information Management

Specialization in
Information and Decision Systems

Universal Genetic Programming: a Meta Learning Approach based on Semantics

Alessandro Re

A thesis submitted in partial fulfillment of the requirements for the degree of Doctor in Information Management

November 2018

NOVA Information Management School
Universidade Nova de Lisboa

NOVA IMS Information Management School

Doctoral Programme in **Information Management**

Prefessor Doutor **Mauro Castelli**, Co-Supervisor

Prefessor Doutor **Leonardo Vanneschi**, Co-Supervisor

© 2018 - Alessandro Re
All rights reserved.

Thesis advisor: Mauro Castelli and Leonardo Vanneschi Alessandro Re

Universal Genetic Programming: a Meta Learning Approach based on Semantics

Abstract

With the advancements of Machine Learning, the number of predictive models that can be used in a given situation has grown incredibly, and scientists willing to use Machine Learning have to spend a significant amount of time in searching, testing and tuning those models. This has an inevitable impact on the research quality. Many scientists are currently working on different approaches to automate this process by devising algorithms that can tune, select or combine multiple models for a specific application. This is the case of ensemble methods, hyper-heuristics and meta-learning algorithms. There have been great progresses in this direction, but typical approaches lack the presence of an unifying structure onto which these ensemble, hyper or meta algorithms are developed. In this thesis we discuss about a new meta-learning method based on Geometric Semantic Genetic Programming. The milestone introduced by this approach is the use of semantics as an intermediate representation to work with models of different nature. We will see how this approach is general and can be applied with any model, in particular we will apply this case to regression problems and we will test our hypotheses by experimental verification over some datasets for real-life problems.

Contents

1 Introduction **11**

1.1 Learning Machines to Solve Hard Problems 11

1.2 Objective of this work 17

1.3 Structure of this work 18

2 Background **19**

2.1 Evolutionary Computation and Genetic Programming . . . 19

2.2 Learning to represent data 29

2.3 Geometric Semantic Genetic Programming 33

2.4 On the Generalization Abilities of GSGP 37

2.5 Ensemble, Meta-learning and Hyper-heuristics Methods . . 40

3 Universal Genetic Programming **44**

3.1 Initializing GSGP with external models 44

3.2 UGP as an Ensemble or Meta-Learning Method 51

3.3 Implementation details 54

4 Experimental Results **58**

4.1 Datasets 58

4.2 Experimental settings 60

4.3 Results 66

5 Conclusion **84**

5.1 Discussion 87

References **99**

Listing of figures

2.1.1 A representation of crossover (XO) and mutation (MUT) operators in Genetic Programming. The crossover operator is often implemented as an operator that swaps two random subtrees of the individuals being processed (in blue and orange); this yields two children, which often take the place of the parents in the population, but some implementation use just one of these two. The mutation operator is often implemented by replacing a random subtree of the individual being processed with a subtree that has been randomly generated (in yellow). 26

2.1.2 A representation of how the evolutionary process works. After initializing the population, the main loop begins with the selection of which individuals that will survive; then, those individuals will be modified by genetic operators (primarily mutation and crossover). The resulting individuals produce a new population that will replace the initial one, and the loop will then continue with the next iteration (generation). 27

2.1.3 In this scheme are represented three, very simple, fitness landscapes. Each node represents a solution, which are sorted on the *y* axis depending on their fitness value. In the leftmost graph, we can observe that there is only one local optima, the topmost node, therefore it is unimodal. In this case, there are no plateaus, because each node can have only neighbors with better or worse fitness. In the central landscape we can observe two local optima, but only one of them is also a global optima. This landscape is therefore not unimodal. On the rightmost figure, we can see that the landscape is unimodal as there is only one local optima, but a space with a similar structure might not be easy to search as it is easily possible to jump among bottom nodes without finding the best one. . 29

2.2.1 Overfitting and underfitting models. A model is said to underfit if it is not complex enough to capture the phenomena to be modeled and, at the opposite, it is said to overfit if it is so complex that learned the data without ability to generalize. 32

2.3.1 Geometric Semantic Mutation is represented. In this case, the difference of two random trees (in yellow and orange) is used to perturb the semantic of the individual being processed. 35

2.3.2 Geometric Semantic Crossover is represented as the weighted sum of the individuals being processed (in blue and orange); a random tree (in yellow) is used to weight each of the individuals to yield a semantic which is geometrically bounded in the sub-region of the semantics of the original trees. . . . 36

2.4.1 Mapping between the problem space (on the left) and the semantic space (on the right). Note how the dimension of the semantic space is defined by the size of the dataset (2, in this case), and not by the dimension of the problem input space (1, in this case). 39

2.4.2 A representation of the semantic of a function $f(x, y)$. The function is usually unknown (thus, it is the objective), but a number of samples are given, therefore we know the values assumed by the function for a given set of input values, $(x_1, y_1), (x_2, y_2), \ldots, (x_m, y_m)$. The set of known values is the semantic of f. The number of samples is crucial to being able to capture, and therefore to approximate, the unknown function (compare orange and yellow samples). 40

3.1.1 To train the models and then select them, the data should be split in 3 partitions: training, validation and testing. We use nested K-fold cross validation. In this representation, data that is used for training is represented in blue, data that is used to validate the models is in yellow and test data to evaluate the overall performances is colored in orange. Each block is a partition p of the data and folds are union of different partitions. 48

4.3.1 Fitness value of the best individual in the population at each generation for the bioav train dataset. These values are averaged for all the 60 runs performed, and reports both the median value that the median absolute dispersion for each curve. In this case, UGP appears to be the best performing algorithm. 65

4.3.2 Fitness value for the best individual in the population for the bioav test dataset. The curves represent the median values of the 60 runs and to each curve is associated its median absolute dispersion. From this plot, we can see that UGP is likely overfitting, while UGPS is performing very well since the beginning of the evolution. 65

4.3.3 These histograms represent the absolute frequencies of the fitness values of the best individuals of the population over the 60 runs performed, for the test bioav dataset. As visible, GSGP suffers from a few outliers, but overall has a good final performance which is comparable to UGPS. 65

4.3.4 These plots represent the Euclidean distance, as evolution proceeds, between the semantic of the best individual at generation 0 with the semantic of the best individual in each other generation. The topmost plot shows the semantic distances for train and test datasets when GSGP is run, while the bottom plot shows the same for the UGPS algorithm. Semantic evolution for UGP is a flat line, due to overfitting, and was not reported. 68
4.3.5 These plots shows the contribution of the models that took part into evolution. In the top one, the contribution is shown for each selected model during the evolution, while below it is shown the histogram of contributions at the last generation. These plots refers to models used by UGPS. In the case of UGP, the only contribution was brought by Decision Trees. 69
4.3.6 This pie chart shows the average running time for the UGPS algorithm. The light gray represents the time required to perform the selection of models, while the remaining part is relative to the long run. These times are averaged for the 60 runs, and it is visible that the strategy we adopted to select the models is clearly requiring a major computational effort. This is due not only to the number of models, but also to the fact that each model was run 20 times in total, due to nested folding cross validation. 70
4.3.7 These plots show the distribution of fitness values for the three algorithms, both for train (above) and test (below) datasets. The represented median values are paired with median absolute dispersion of the data. As it happened with the bioav dataset, we can witness again to an overfitting phenomena. . 71
4.3.8 Above: the contribution, during the evolution of the UGP algorithm, of all the models. Below, the same plot, but for UGPS. Note how in UGP the contribution values are much less noisy than the ones in UGPS. We attribute this fact to Decision Trees being an overfitting solution which has been discarded by the selection procedure in UGPS. 72

4.3.9 This histogram shows that some combination of models were selected more frequently by PESP, but note there is not a prevalent group. Interestingly, Decision Trees are never selected, but Random Forest are always selected, suggesting that the former might lead to overfitting solutions. 73
4.3.10 This figure shows evolution of semantic distance between the best individuals. Note that there is a progress in the evolution of the semantic, even if the initial solution turned out to be very good since the inception. We attribute this fact to the ability of GSGP to escape local optima and improve the solutions. 73
4.3.11 In these plots, we can see how UGP and UGPS perform against GSGP in terms of fitness values of the best individuals during the evolution. The new algorithms are performing very well on training data, scoring almost zero error, and are still significantly better than GSGP on the test dataset. 75
4.3.12 Above, the plot shows the evolution of the fitness values of the best individuals when using UGP. Below, the same plot for UGPS. Note how the train and test values are stable, without increasing or decreasing over time. This likely means that the Base Learners found an optimal solution and that GSGP is not able to improve upon it, not even by overfitting. 76
4.3.13 In this figure, plots of distance evolution of GSGP (above) and UGPS (below) are compared. Note how, while GSGP is actually working on the transformation of semantic values over the evolution, UGPS has never changed the semantic of the best individual. This is likely due to the elitism mechanism, which kept a solution from a basic learner into its place, with no chances to alter it. This is due to the fact that such solution is an excellent result, possibly not far from being a global optima. 77

4.3.14 These plots show fitness values of the best individuals during the evolution. As visible in the plot above, the fitness value in UGP and UGPS is almost zero, meaning that the algorithm can model the training data almost with no errors. Below, we can see that the error on test data is not zero, as expected, but it is not a case of overfitness, as the algorithms are performing much better than GSGP, even if evolution did not bring any significant contribution in the final results. 79
4.3.15 In this plot it is visible the evolution of the semantic distance between the best individuals at each generation since the first generation. Note how, both in UGP and UGPS, above and below, respectively, evolution could slowly change the semantics. This appears to be a case in which GSGP escaped a local optima and could find another area to exploit in the fitness landscape. Interesting to note how the semantic distance, albeit on a very small scale, has its major shift in the test dataset only. 80
4.3.16 This plot shows the contribution brought by selected models during UGPS evolution. Note how Genetic Programming and Decision Trees are the main contributions, strictly tied to each other. Random Forest scores also a relevant contribution during the evolution. 81
4.3.17 This histogram shows the average relative contribution of each model at the last generation of UGP. In this case, we can see that Decision Trees and Genetic Programming are contributing almost exclusively to the evolution, altgough Random Forest has also a minor role in this process, influencing about 7% of the final best solution. 82

1 Introduction

In this chapter we will see what are the motivations of this work and which problems it tries to solve. We will overview the concepts and the ideas that are involved.

1.1 Learning Machines to Solve Hard Problems

Some tasks are very easy for humans

As humans, we daily face a high number of problems which are not clearly defined and for which the process that involves the identification of a solution is very hard to define. For some of these problems we have a natural talent and we can solve them quickly and effortlessly: recognizing a friend in a crowded place, walking to her without bumping in other moving people, talking and hearing her voice over the noise of other people chatting and, of course, understanding what she says and having a decent conversation are tasks which we can solve without even realizing how complex they are. Although we might be (partially) hard-wired to solve this kind of problems (for example, a baby can learn any language independently from which language and the baby's

origins[82]), we are not born with the knowledge to solve these problems, and we develop the required skill naturally, in our infancy.

Tasks easy for humans might be very hard to formalize and resolve algorithmically

Programming a computer to solve the problems listed above is an extremely difficult task with current technologies, mostly because the problems are not precisely defined on multiple levels: the task of "identifying a specific friend in a crowd" might be easy to understand or formulate, but an in-depth specification of how such a task can be carried is very hard to define. A typical individual can indeed recognize a friend in a crowd, but is absolutely unable to explain how he can do that: it just happens and the inner working, that is, how the brain works, is still mostly a mystery, a black box.

Machine Learning is a way to solve hard tasks with experience

Machine Learning (from now on, ML) is one possible attempt to solve this kind of problems automatically using computers. Mimicking humans, the idea behind ML is to learn from data and experience: observe data to extract interesting patterns, summarize knowledge and devise an automatic procedure that can use that knowledge to solve specific tasks. To state it more rigorously, with the words of Tom Mitchell[70]:

> A computer program is said to **learn** from experience E with respect to some class of task T and performance P, if its performance at tasks in T, as measured by P, improves with experience E.

Typical learning tasks are supervised, unsupervised or by reinforcement

The learning process is usually iterative and can be formulated in different ways: do we have a teacher or not? Can we perform actions in the environment, or not? Do we know the correct answers to the questions we might have on the problem or shall we figure them out by ourselves? Is the environment deterministic or stochastic? Depending on the answer to some of these questions, we can group ML techniques in different categories. One diffused categorization divides approaches in 3 groups: supervised learning, in which we are given a set of observations, a question on these observations and a correct answer to every of these questions; unsupervised learning, in which we are given observations and a question, but no knowledge regarding the answer; reinforcement learning, in which we can act in an environment, collect observations following our actions and in which a teacher may, from time to time, provide feedback to our behavior toward a given objective. More formally, in

supervised learning, we are given a dataset of examples comprising both input to the program and the expected output, $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^{N}$, where N is the size of the dataset (number of examples), x_i is the i-th input and y_i is the i-th output (also known as response variable and label). If the output is continuous or from an infinite set, for instance $y_i \in \mathbb{R}$, we talk about **regression** problems, otherwise, when it is discrete, categorical or nominal, we talk about **classification** problems[73]. In the case of learning from data which is not labeled, the task is usually to learn a description of the data. In this case, we talk about unsupervised learning or knowledge discovery, and we have $\mathcal{D} = \{x_i\}_{i=1}^{N}$. Finally, when we are given partial information in response to our actions, we talk about reinforcement learning, in which the objective is usually to learn a behavior, or a policy, that guides our actions inside an environment[95]. In this thesis, we deal exclusively with solving regression problems in a supervised learning context.

The concept of machines able to learn is as old as computers themselves

Being learning a fundamental activity in all human individuals and our society, it is not surprising that the idea of machines that could learn is rather old in the history of computer science. In fact, Alan Turing himself hypothesized the idea of Machine Learning:

> If the machine were able in some way to 'learn by experience' it would be much more impressive. If this were the case there seems to be no real reason why one should not start from a comparatively simple machine, and, by subjecting it to a suitable range if 'experience' transform it into one which was much more elaborate, and was able to deal with a far greater range of contingencies[98].

The concepts that he formulated are fundamental in the ML community: it is in fact important that not only a machine is capable of learning, but it is also able to apply its knowledge in a wider set of situations. When this happens, we say that the machine is able to generalize.

First machine learning models to solve games or model the brain

Since him, pioneers devised a variety of ways to let machines learn, for instance by using statistical or probabilistic techniques to infer parameters of a model, or by producing computer programs which can write computer programs. Arthur Samuel pioneered the field by coining the term Machine Learning and exploring the application of learning problems by developing a program that played checkers and improved its performances after playing against

itself[90]. Other pioneering approaches include, for example, the work of McCulloch and Pitt who, in 1943, introduced a computational model of the neuron, a single cell of the brain[64]. This funding work opened the path to the field of research later known as connectionism.

Natural evolution was a source of inspiration for solving computational problems

More relevant to this thesis are the works in the field of Evolutionary Computation. The idea can be traced back to Turing who, in 1948, proposed the use of genetic or evolutionary search, with genes combination and survival as goodness measure[97]. It took more than a decade to have advancements on this front and, during a short time-span in the 1960s, different research groups come up with approached inspired to natural evolution. Evolutionary Programming, Evolution Strategies, and Genetic Algorithms were introduced[29, 37, 86], opening the path to new research fields. In the last decade of the 20th century, a new research field born, namely Genetic Programming[44, 45], and all of them converged in the field of Evolutionary Computation.

Individuals that are fitter in an environment survive and reproduce more

As evident from their names, the underlying principle is to borrow the principles of Darwinian evolutionary theory to solve problems. In this theory, individuals compete in a given environment with a limited set of resources. The individual that fits best in its environment (which is not necessarily the strongest or the fastest) will survive more, therefore will have more chances to reproduce and to generate more offspring. Newborn individuals are subject to small random changes from their parents that will, over time and thanks to the selection mechanism, lead in significant shifts in how the individuals compete and are fit in their environment[23]. The environment is very important to determine how individuals will evolve: the fitness of one individual is not one intrinsic property, but is something defined with respect to it. The differentiation among individuals is due to the differences in their genetic code, or genotype, which in nature is a long and complex protein that determines the physical appearance of the individual and its abilities to interact with the environment and other individuals, this is called the phenotype.

Evolutionary Computation is based on a loop of evaluation and modification of individuals

Each evolutionary algorithm is based on different ideas and interpretation of these principles, but the fundamental idea is that individuals can represent a solution in an optimization problem to be solved and that evolution is a mean by which solutions are modified until they are satisfying. Using evolution to solve this kind of problems usually requires defining what a solution is, how

to represent it and how to measure its fitness, that is, its performances when compared to other individuals.

A motivation for using EC is that evolution is effective

The adoption of an evolutionary approach to solving problems is justified by the fact that evolution is the recognized principle that allowed the existence of humans and a plethora of living beings that are extremely efficient and competitive in their environments. It is a very general method that allows to freely adopt any appropriate representation for a given problem and it is often surprising in its ability to find unexpected and effective solutions [46, 54].

Evolutionary Computation can be used in Machine Learning as well

Evolutionary Computation involves an iterative process that produces, by successive refinements, a solution to a problem that is hopefully acceptable. ML and optimization techniques are strictly interconnected: to find a model which is able to generalize it is necessary to solve an optimization problem, for instance that minimizes a given error over known data. The ability to generalize, and therefore having a model which really has "learned" the essence of the problem under some sense, usually requires additional steps and care. Therefore, Evolutionary Computation can be used to solve problems using ML, and with time and experience we can see that it is also a quite effective technique.

Machine Learning has a major role in our current lives

In the years, Machine Learning acquired a major role in the tools we build and use in our daily lives, and it is very hard for an individual in our society to not using it, directly or indirectly. For example, automatic learning is used in manufacturing, in production, dispatch or selling of goods and, of course, in the service industry.

There are many ML models and making a choice is not always easy

The role of ML is especially of particular importance for every person that has to make sense of (many) data, and the scientific community is not excluded. Indeed, the level of literacy regarding coding and software development is increasing in the years, and so it is the knowledge regarding ML. Nowadays it is very common to run algorithms that learn from data to perform basic tasks such as regression, classification, and clustering: it often gives insights on the data and improves the quality of the results for the entire scientific community. Nevertheless, the pace of progress in the ML community and the number of models available is overwhelming and a regular scientist does not always have time to learn properly how each algorithm works, how to use and how to tune it. Therefore, it is not surprising that a small number of simple (to use and to understand) algorithms are being widely used for a multitude

of tasks. These algorithms have the merit of performing well in general, to be conceptually easy to understand and, therefore, to debug and use.

Determining what model to use for a problem is a problem itself

This, however, limits the progress of the entire community. Naturally, an important piece of the puzzle is the role of software engineers and data scientists that can abstract from the "low level" of some algorithms and make them freely accessible to the general scientific community, but often this requires careful engineering and design of tools. This effort, although very useful in general, does not eliminate the need of searching for better models: the general methods will not work in every case, and some other approaches might be more suitable to solve a given problem. Typically, a scientist will evaluate the performances of different ML algorithms over the same problem and use the one that appears to work best.

In his book "The Master Algorithm", Pedro Domingos discusses an algorithm that is capable of solving a plethora of different problems[25]. The research community is working hard towards that goal, but it is still far. He highlights how few algorithms are currently used in a very large variety of applications: the key is naturally their generalization abilities, not necessarily as Machine Learning models, but also as useful algorithms with widespread applications. Evolutionary Algorithms are, in fact, one of the selected candidates to become master algorithm: they are indeed very powerful and generic enough to be run many different applications. But, even if they can be applied to many different domains, careful setup and tuning is required to obtain significant performances and qualitative output. An unifying, higher level algorithm is still required to make these powerful algorithms more usable.

Many paths are being explored in the attempt to automatically use the best model

Numerous approaches are being explored, included but not limited to transfer learning, ensemble methods, hyper-heuristics, and meta-learning. The former is the practice of learning from one experience and transfer the experience to solve new problems. This practice is diffused in the connectionism field. Ensemble methods, on the other hand, try to solve a problem by using a number of models, similar or different among themselves, and selecting or combining them to yield performances which are better than the constituting parts. Also hyper-heuristics and meta-learning, which have many similarities among them, try to solve a problem by employing a multi-tier approach in which one algorithm tunes or builds one or more algorithms to solve a given

problem. These methods work by working on a more abstract level, for example by selecting which underlying algorithm to use for a given problem or by tuning the (hyper) parameters of the underlying algorithm. We will discuss these approaches in the next section.

We think a middle-level abstract could help in automating this task

What we think is still missing to these approaches is the presence of a conceptual middle level which creates an abstraction layer: while it is true that we can combine multiple models in an ensemble method, it is also true that how we combine them greatly depends on what kind of models they are. The same can be said for hyper-heuristics and meta-learning: although it is indeed possible to use an algorithm to tune the parameters of another algorithm, the former must be designed so that it knows how to work with those parameters and this requires a knowledge of the algorithm itself. This approach can be very useful and achieve great results, but it is not universal.

1.2 Objective of this work

Our objective: build an ensembling algorithm using an intermediate representation

For the reasons introduced above, in this thesis, we tackle the problem of building an unifying algorithm which uses an "intermediate representation" to abstract from the model actually used and, therefore, is extremely general. We will see how recent advancements in the field of Genetic Programming enabled a new way of searching for problem solutions and how this might impact the Machine Learning community in general.

Geometry Semantic Genetic Programming is a good starting point thanks to how it uses semantic

In particular, we will build upon the results obtained by the Geometric Semantic Genetic Programming community, in which the concept of semantic is key to build the necessary abstraction. In this context, the semantics of a program has the same meaning of the image of a function in mathematics: given a set of input data, the semantic of a program is the value of that program on each of those data. However, we will usually deal with data in tabular form, therefore the semantics of a program will be a vector instead of a set.

Semantic can be a valid intermediate representation for any ML model

We will see how this approach is flexible in the sense that it can be used with any underlying model, as long as the output of such a model is meaningful. We will discuss the principles and the algorithms in detail, showing how the algorithm we are introducing will have to face the same problems that are to be faced by Machine Learning practitioners when they look for a given model.

We will suggest a solution to these problems and provide an implementation of the algorithms. Then, we will test these implementations on five different datasets, comparing the obtained results with the results obtained in the literature.

1.3 Structure of this work

This thesis is structured as follows. In Chapter 2 we provide the necessary elements required to contextualize and understand this work; in it, we will introduce the concepts of Genetic Programming, Geometric Semantic Genetic Programming and Ensemble Methods, seeing how the current state of the art could be advanced to exploit even more the concept of semantic. In Chapter 3, we will introduce the main contribution of this work: two variations of Geometric Semantic Genetic Programming called Universal Genetic Programming with and without pre-evolutionary selection. In Chapter 4 we will show the tests performed over five datasets to support the utility of the algorithms and discuss their results. Finally, in Chapter 5 we will discuss the outcome, the flaws and the qualities of the proposed algorithms and the obtained results, suggesting possible improvements and new paths for future research.

2 Background

In this chapter we lay out the foundations over which this thesis is developed. We will introduce Evolutionary Computation and Genetic Programming. We will introduce the concept of semantic of a program and introduce Geometric Semantic Genetic Programming, which is built upon Genetic Programming and onto which we will build upon. We will also introduce three classes of methods that try to improve our efficiency in using Machine Learning models by combining and tuning them automatically. These are Ensemble, Hyper Heuristics and Meta Learning methods. They are similar in their intents, but each one presents some key differences from the others that we will introduce.

2.1 Evolutionary Computation and Genetic Programming

GP is an algorithm that is able to create programs using evolution

Genetic Programming (from now on, GP) is an active and exciting research sub-field of Evolutionary Computation [27]. The main characteristic of GP is that it can be used to solve optimization problems by producing computer programs through the principles of natural evolution. In particular, GP uses

an iterative process which involves refining randomly generated programs by selecting, reproducing, combining and modifying them until the desired level of approximation is reached on the problem to be solved.

GP is a very flexible optimization algorithm with multiple application domains

GP has been used to solve a variety of problems, for example to generate art [22, 59, 88], to design satellite antennas [56], to play chess [36], in game theory [20], to develop RoboCup players [58], for forecasting of complex phenomena (crime rates [19] or ozone concentration [17]), quantum computing [94], symbolic regression [11, 44], sorting[41], signal processing [61] and in many tasks it performed very well, producing human-competitive results[46].

We will now review the fundamental concepts in GP, but for a more broad overview we suggest [5, 44, 45, 53, 83]. We will start to analyze GP from a specific case, known as Standard GP [44].

Optimization problems as search in a solution space $\mathcal{S}$

To set up an effective evolutionary process that finds solutions to a problem, it is first necessary to formalize and set a representation for the solutions to such a problem. In general, given a set $\mathcal{S}$ of solutions, an optimization problem consists in finding one or more $s \in \mathcal{S}$ such that they satisfy some desirable properties. To quantify how much a solution is desirable, a function $f : \mathcal{S} \mapsto \mathbb{R}$ is usually employed to measure the performances of such solutions. If a better solution has a higher value of f, the problem is said to be a maximization problem, otherwise it is a minimization problem.

Different representations might impact the efficiency of the search

How solutions in $\mathcal{S}$ are represented is fundamental to solve efficiently a problem [76, 89]. For example, given an instance of the classic Traveling Salesperson Problem (TSP), in which the input is a graph describing the distances between the places to visit, a solution could be represented as a sequence of places to visit, with no repetitions, or as a program written in C that reads the graph as input and computes the length of the shortest path. In the first case, the space of solutions $\mathcal{S}$ corresponds to the set of permutations of the cities, and evaluating a solution is done by summing the distance between subsequent pairs of cities (that is, f contains the knowledge necessary to interpret the sequence and to produce an output value), but in the second case, $\mathcal{S}$ is the set of programs that can read the graph and return a valid real number, which is a much larger space (consider, for example, that there are multiple algorithms that can be used to solve the same problem). As can be seen, while the first representation looks naturally efficient to solve the TSP problem (although,

we are not stating that it is *the most* efficient representation), the latter could be suitable to solve a class of problems as it can process any graph. This difference is what makes GP different from other evolution algorithms such as Genetic Algorithms (GA), in which the representation of a solution shall be, at least in its original formulation, a linear sequence of fixed length. Since in GP there is no such restriction, it could be argued that GP is a class of algorithms that include Genetic Algorithms as well [49].

How solution are represented heavily influences our ability to search in $\mathcal{S}$

Even if the space of solutions $\mathcal{S}$ is not the only factor influencing the difficulty of solving a problem with GP, in practical terms, it is often a primary concern and for this reason the selection of a representation is crucial for solving a problem [30]. In the following discussion, we will see how the structure of f and $\mathcal{S}$ has a role in the difficulty of a problem. Selecting a representation is very important, not only by impacting the search efficiency, but determining if an optimal solution can be found at all.

Standard GP uses trees to represent usual arithmetic expressions

In Standard GP, solutions are often represented as arithmetic expressions composed by the usual operators: addition, subtraction, multiplication and division, defined on real numbers. To efficiently represent these expressions, operators are combined into a tree, in which internal nodes are the operands and leaves are scalars. Those scalars can be numerical constants, such as π or 42, or input variables, such as x_0 or x_{12}, with x_i being the i-th component of a given input vector $\mathbf{x}$

Many different representations have been used in literature

Although GP imposes very few constraints on the kind of solutions it can work with, we will deal mostly with solutions represented as trees, as we just described, hence we will discuss them in detail. Nevertheless, it is not required to adopt expression trees, and many other representations have been used successfully including, for instance, lists [7], assembly languages [21], stacks [80, 93], grammars [75] or binary space partitions [84].

Terminals are leaves of the tree, internal nodes are functionals with various arity

In the context of Standard GP, as well as in our case, trees are composed by nodes which are either functionals or terminals. Functional nodes are nodes that accept inputs and produce a single output, while terminals are nodes that accept no inputs. Functionals with a single input, such as the absolute value function $|\cdot|$, are said to be un*ary*. On the other hand, functionals with two inputs, such as the sum function $\cdot + \cdot$, are said to be bin*ary* operators. In general, the *arity* of one operator determines how many inputs it accepts. Terminal

nodes are either constants, variables or nullary operators, while functionals have arity greater than zero.

Programs can use one or many data types in a weakly or strongly typed fashion

Until now, we assumed that functionals and terminals were real-valued, but it is not necessarily so: the definition of different types is an important topic in computer science, and GP is no exception. Although strongly, multi-typed GP systems exist [71], in which it is possible that multiple data types coexist in the same evolutionary process. In our case we impose that every node in the tree assumes a value of a single type, that is a real number, as it is common when defining usual arithmetic expressions. Therefore, our functionals are closed under their type, meaning that they accept parameters of the same type they assume. to keep the individuals in a consistent state. Note that the tree structure imposes a precedence on the operators, as it is usually evaluated recursively from leaves to its root, therefore it is not necessary to have parentheses.

Error handling is an important part of automatic program production

It is also important to define how errors and exceptions are handled. For example, the division operator is usually not defined over zero and trying to perform a zero division will likely cause a runtime error in the system where GP is programmed. Similar argument can be made for any function with limited domain: for example, logarithms cannot have negative argument. Therefore, to avoid errors, specific functions are defined (for example, the division operator will be defined in 0, for example by assuming value 1), or constraints on data are imposed, so that it is possible to compute an expression without compromising the overall evolution.

To increase the effectiveness of a GP system, Koza suggested to select functionals which are closed, that is, they are both type consistent and safe to evaluate [44].

The evolutionary process acts upon a population of individuals (solution)

Once a representation has been set, the evolutionary process begins by considering a set, usually constant in size, of solutions. Borrowing from biology, solutions are often named individuals and the set of individuals is called population. Such a set is initialized by an initialization operator, which adopt a certain initialization strategy to define which individuals will compose the initial population. In Standard GP, where trees are used to represent expressions, there are three common ways to initialize individuals: grow strategy, full strategy and ramped half-and-half.

Trees are built with different strategies such as, grow, full and ramped half-and-half

The grow strategy consist in picking a random node among the entire set of functionals and terminals: if a functional is selected, it is used as a root, and the grow strategy is applied recursively to create each of its nodes. The procedure ends when a terminal node or a maximum depth is reached: in this latter case, selection over terminal nodes is enforced at the maximum height. In the full strategy, full trees are created by randomly picking functional nodes until the last but one level is reached, at that point, only terminal nodes can be selected as leaves. In the ramped half-and-half, both grow and full strategies are used to initialize half of the population each, with the constraint that instead of using a single maximum length, a range of heights is used so that different fractions of the population will be initialized using a different maximum height. This ensures a higher degree of diversity within the initialized population. Ramped half-and-half is a widely used initialization strategy and we will use it in this work as well.

GP uses an iterative process to evolve the population

Before the iterative process begins, the initialization operator will create individuals according to the selected strategy. In the context of Standard GP, every individual is initialized accordingly the chosen strategy, yielding a population of randomly generated trees (expressions). Once initialization has been performed, the iterative evolutionary process will start. Every iteration of this loop will be referred as generation, and as it happens with natural evolution, each generation of individuals is the result of natural selection applied to the previous generation.

Selection, reproduction, recombination and mutation are the basic operators in GP

In nature, during a generation, individuals are first selected, then they can reproduce and usually some amount of recombination and mutation will happen to their DNA, via sexuated reproduction and environmental influence. In GP, this process is performed by the selection, crossover and mutation operators. Selection operators require that the performances of each individual have been computed: each individual is, in fact, a solution of the optimization problem we are trying to solve and after evaluating it (e.g. by compiling and running it) over the problem inputs, we can measure the quality of the outputs and determine its fitness, that is, how fit the individual is in its environment and how likely it is to reproduce. The higher the fitness, the likely one individual will reproduce.

The fitness function determines the goodness of each individual

Generally speaking, the fitness of one individual is a quantity that we want

to maximize: more fit means a better individual in its environment. In the field of Evolutionary Computation, the meaning of fitness can often be different from the biological term, and fitness function is often interpreted as an objective function, that is, f. Therefore, for maximization problems, the objective is to search solutions $s \in \mathcal{S}$ that yield the highest value in f, while for minimization problems, we seek solutions that yield the lowest possible value of the fitness function. We will adopt this convention where fitness function and objective function are the same entity, and we will restrain from referring to a specific minimization or maximization problem by speaking, more generally, to a better or worse fitness value. In minimization problems, better means smaller, while it means larger in maximization problems.

Selection operators use fitness to determine which individuals will survive one generation

The fitness of each individual is therefore used by the selection operator, which randomly picks every individual with a probability that is correlated to its fitness. In other words, the better the fitness, the higher the probability an individual will be selected. As with initialization, there are different strategies to select individuals. Common strategies are Tournament Selection [6, 68] and Proportional Selection [31]. In the Tournament Selection, a fixed number of individuals (the tournament size) is randomly selected with uniform distribution and, only among those individuals, the fittest individual is selected as tournament winner and it can, therefore, advance from one generation to the next (i.e. survives the natural selection). Another widely used strategy is the proportional Selection, also known as Roulette Wheel selection, in which each individual has a probability to be selected for reproduction with probability proportional to its fitness. Note how, in this case, fitness must be computed for every individual, while it is this does not happen for the Tournament Selection.

Selection pressure reveals how much individuals are competitive in surviving

A property of selection strategies is their selection pressure, which is a measure of how the system is biased towards fitter individuals: if the selection pressure is high, individuals with better fitness will have a much higher probability of being selected, while if selection pressure is low, all individuals are likely to be selected independently from their fitness. For example, if the tournament size is equal to 1, selection pressure for the Tournament Selection is at its minimum and every individual will be selected with uniform probability (any selected individual is also the winner of its tournament), while if the

tournament size is large, only the fittest individual will be selected, therefore the fittest individuals in the population will likely survive. It is obvious that, if the population size is constant from one generation to the next, selection is meaningful only if individuals are selected with replacement, otherwise all individuals would be selected and survive each generation. The ability of one individual to be selected multiple times, roughly correspond to the ability of a living being to reproduce more. We illustrated tournament selection as it is the strategy we are using in this work, but many more selection schemes that have been studied and adopted in literature. For a more in-depth analysis, the interested reader can refer to [31, 63, 67].

Crossover swaps genetic material, while mutation introduces genetic novelty

Once selection is completed, individuals might be present in multiple copies and are therefore reproduced. To these individuals are then applied the recombination and mutation operators [57, 102]. Recombination operator, often called crossover, intuitively works by swapping genetic material between two individuals. More precisely, it combines the representations of two or more "parent" individuals to produce one or more "children" individuals that are built with fragments of the parents. Note that no new genetic material is introduced, but the existing material is recombined. In the context of Standard GP, two parents can be crossed over to produce two children by randomly swapping elements from one parent to the other. There are, as usual, multiple strategies for doing this, but the one most adopted is to swap two random subtrees, which can be selected completely randomly, or with some constraints (e.g. never pick root nodes, or pick random subtrees so that, after swapping, the resulting trees do not exceed a maximum height limit). On the other hand, mutation, which is represented along with crossover in Figure 2.1.1, is the operator that introduces new genetic material. Mutation is fundamental to explore the entire solution space, because with its absence, the available genetic material would be determined at initialization (e.g. if no expression with a sum is generated, then no expression can have a sum in it, which might be fundamental to reach optimal solutions). Mutation works on a single individual and changes it in a random fashion and, as with every other operator, there are plenty of strategies that can be applied. A common strategy, when working with trees in Standard GP, is the Subtree Mutation, which consists in randomly picking a node in a tree and replacing it with a randomly

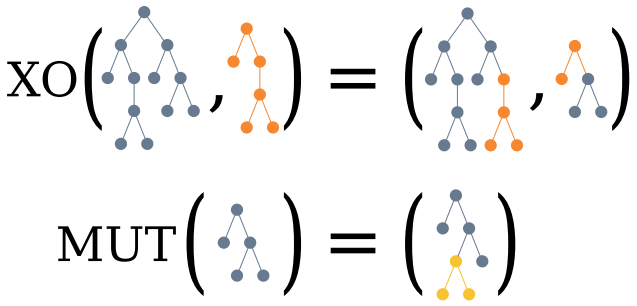

Figure 2.1.1: A representation of crossover (XO) and mutation (MUT) operators in Genetic Programming. The crossover operator is often implemented as an operator that swaps two random subtrees of the individuals being processed (in blue and orange); this yields two children, which often take the place of the parents in the population, but some implementation use just one of these two. The mutation operator is often implemented by replacing a random subtree of the individual being processed with a subtree that has been randomly generated (in yellow).

generated subtree (possibly limiting the height of the generated tree, to satisfy height constraints for the resulting tree). Another common mutation is Point Mutation, which randomly picks a node in the tree and replaces it with a random node with the same arity (i.e. a terminal with a terminal, or a functional with a functional with the same number of arguments) [83].

The iterative process continues until a termination condition is met, for example, the evolution stops after a fixed number of generations, or when the fitness of the best individual in the population reaches a fixed threshold. A representation of the algorithm is shown in Figure 2.1.2.

Elitism preserves the best individual from one generation to the next

It can be seen from this explanation that there is no guarantee that a good solution, once found, survives the selection mechanism. In fact, we are usually interested in finding the best possible solution encountered in the entire evolution process, not the best one at the last generation. Elitism is a strategy used by GP to guarantee that the best individual will survive the selection process. Usually it means that the fittest individual is automatically included in the population of the next generation, without undergoing any selection procedure. This means, of course, that the fitness must be evaluated for every individual at every iteration, but employing elitism is often a good strategy as it helps to improve the overall fitness of the population.

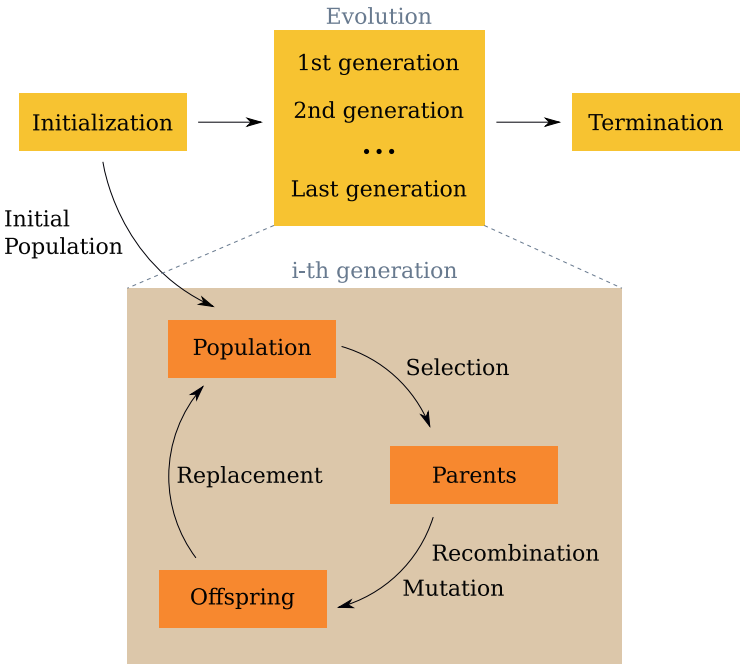

Figure 2.1.2: A representation of how the evolutionary process works. After initializing the population, the main loop begins with the selection of which individuals that will survive; then, those individuals will be modified by genetic operators (primarily mutation and crossover). The resulting individuals produce a new population that will replace the initial one, and the loop will then continue with the next iteration (generation).

Useful when reasoning on GP are the concepts of neighborhood and of fitness landscape. The neighborhood of one individual is the set of solutions that can be obtained by transforming it according to one application of recombination or mutation operators. If the fitness of an individual is better than the fitness of all its neighbors, then the individual is a local optimum. A global optimum is the best individual among all local optima. Note that multiple global optima might exist, but their fitness value will be the same.

The concept of neighborhood provides a structure over the space of solution

The concept of fitness landscape is based on the idea of neighborhood, but gives an insight on the difficulty of a problem by mapping the structure of the individuals and their neighborhood in a geometric space [76]. There are different definitions, all roughly equivalent, of fitness landscape, but in simple terms, it is a plot where solutions are placed on the horizontal direction (as independent variables) and their fitness is placed on the vertical direction (as dependent variable) [52]. If the solutions can be represented in a 2-dimensional space, consistently placing the neighborhood structure on the axis, the fitness landscape can be seen as a 3-dimensional map where tops of hills represent local optima.

The structure of the neighborhood defines the fitness landscape of the problem

Another view of the fitness landscape, alternative to the continuous "map", can be obtained by drawing a graph in which solutions are nodes and the neighborhood relationship is represented by edges. This graph places nodes sorted vertically according to their fitness, with nodes with the same fitness on the same level. Local optima are nodes whose adjacent vertices are positioned in lower levels. The topmost nodes are the global optima. This representation has the advantage of not requiring to sort solution on multiple axes according to their neighborhood structure, which translates to an easier visualization which does not require to place solutions in a high dimensional space. A simplified representation of some fitness landscapes can be seen in Figure 2.1.3

Fitness landscape as a 2D graph in which nodes are sorted along one axis according to their fitness

From the fitness landscape it is possible to understand the difficulty of the problem by considering how many local optima there are [42, 69]. Simplifying, the less local optima, the better, and the less plateaus, the better. To understand this intuitively, consider picking a valid, random solution. If the solution is a local optimum, it means that none of its neighbors are better than it, therefore no mutation or crossover is sufficient to find a better solution in a single step. Naturally, since the selection mechanism is stochastic, it is possi-

The difficulty of a problem is sensibly influenced by the fitness landscape

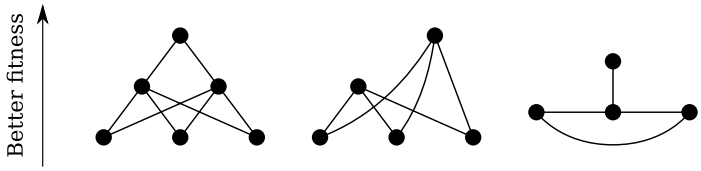

Figure 2.1.3: In this scheme are represented three, very simple, fitness landscapes. Each node represents a solution, which are sorted on the y axis depending on their fitness value. In the leftmost graph, we can observe that there is only one local optima, the topmost node, therefore it is unimodal. In this case, there are no plateaus, because each node can have only neighbors with better or worse fitness. In the central landscape we can observe two local optima, but only one of them is also a global optima. This landscape is therefore not unimodal. On the rightmost figure, we can see that the landscape is unimodal as there is only one local optima, but a space with a similar structure might not be easy to search as it is easily possible to jump among bottom nodes without finding the best one.

ble to escape a local optimum if the peer pressure is not excessively high. On the other hand, if a single global optimum is present and if every solution, but the local optimum, has at least one better solution, it is easier to find the global optimum, as the likelihood of improving is much higher due to the fact that from every solution it is not required to backtrack to worse fitting solutions to reach a better one. A landscape in which there is only one local optimum is called unimodal, otherwise it is called multimodal.

Unimodal fitness landscapes are often desirable

As we explained, having a unimodal fitness landscape is usually a desirable property, but not every fitness landscape that is unimodal is easy to search. As an example, consider a fitness solution which assumes value 1 when a specific solution is found and 0 in every other point. The associated fitness landscape will have a single spike, connected to all its neighbors, and will be a plateau in every other point. In this case, optimization is very hard for many common neighborhood structures.

2.2 Learning to represent data

It is not always possible to encode the modeled phenomena in the fitness therefore we can use a data-driven approach

In the Standard GP that we illustrated earlier it is possible to identify the space of solutions S, composed by all the trees that form valid expressions and

for which a fitness value is defined. The fitness function f maps points in the solution space to fitness values. How the fitness is defined depends on the nature of the problem to be solved, but in general we described it as a function that maps inputs to outputs. Outputs are just a goodness measure for individuals, and do not have to map directly to the problem. Ideally, the fitness function encodes how the solution maps to the problem and then evaluates its performances. For example, if the space of solutions was the space of all possible programs that safely drive a car, the fitness function would measure how safely and how far the program would drive: the more miles driven with no crashes, the more predictable is the behavior by other drivers and the less rules infringed, the better. The fitness function would probably evaluate the solution program by simulating a virtual environment and eventually would compute a value to determine how good is the program in driving. On the other hand, other problems cannot be solved in this way. Consider, for example, the problem of determining if a molecule is toxic or not to humans: we are currently unable to estimate the effect of a certain amount of such molecule by simulating the entire human body, therefore we have to approximate the estimation even more and use simpler models. We can abstract this process and assume that there is a function that, given the description of the molecule being studied, can determine its toxicity level on the average human body. It is naturally very hard to explicitly define this function, but we could try to approximate its extension by sampling some of its values by actual measures. In other words, we can give a safe amount of this molecule to human subjects and measure the toxic effects. By collecting more data, we can better approximate the behavior of such function. Naturally, this does not solve the problem we had, but now we can try to approximate the function by fitting the known points.

Using collected data to model a problem is a key practice in ML

Many problems cannot be simulated, or are excessively expensive to simulate, therefore this data-driven approach is very diffused. In the field of Machine Learning, the whole class of supervised learning is devoted to trying to model an unknown function which maps known inputs to known outputs, which are often obtained by external human intervention. As an example, consider the famous problem in the context of supervised learning is the handwriting recognition problem, in which the known data are a set of raster images

representing handwritten symbols, with the objective of automatically determine what symbol is depicted. This is done using a dataset in which each digit image is labeled. For example, in the very famous MNIST dataset, there is a collection of images of handwritten digits from 0 to 9.

It is worth noting that these problems are often divided into two categories: regression or classification. A regression problem is a problem in which the objective function assumes values in a continuous space, while classification is when outputs assume discrete values. In this thesis, we will deal with regression problems.

We organize data in a tabular form representing the mapping of input variables to one output

From now on, we will assume that data is available in a tabular form, in which a dataset $\mathcal{D}$ is a non-empty list of samples of an unknown function $\hat{f} : \mathbb{R}^n \mapsto \mathbb{R}$. Each sample is a pair $(X, \hat{y})$ where $\hat{f}(X) = \hat{y}$, and such dataset basically represents all the knowledge we have on the problem to be solved. Our objective is to identify a function f which is as similar as possible to $\hat{f}$, given the dataset.

Performances of a model over some data are measured using an error measure like MSE

When searching the approximated function f, we will measure the performance of a candidate function by evaluating it on the dataset and measure how distant is the result from the expected output. The distance measure may vary, but the ones commonly used are the Mean Square Error (MSE) and Mean Absolute Error (MAE), which we will use in this thesis. The MSE is defined as follows:

$$MSE = \frac{1}{N} \sum_{i \in Q} (t_i - y_i)^2$$

where y_i is the output of a candidate solution on the $i - th$ training case and t_i is the target value for that training case. N denotes the number of samples in the training or testing subset, and Q contains the indices of that set. By using the same notation, the definition of the MAE is the following:

$$MAE = \frac{1}{N} \sum_{i \in Q} \|t_i - y_i\|$$

.

Naturally, a function f may behave well on the dataset points, but misbehave on every other point. In this case, we say the function f is unable to generalize well, and in particular that the solution is overfitting. A graphical representation of overfitting and underfitting (i.e., the opposite phenomenon) is

displayed in Figure 2.2.1.

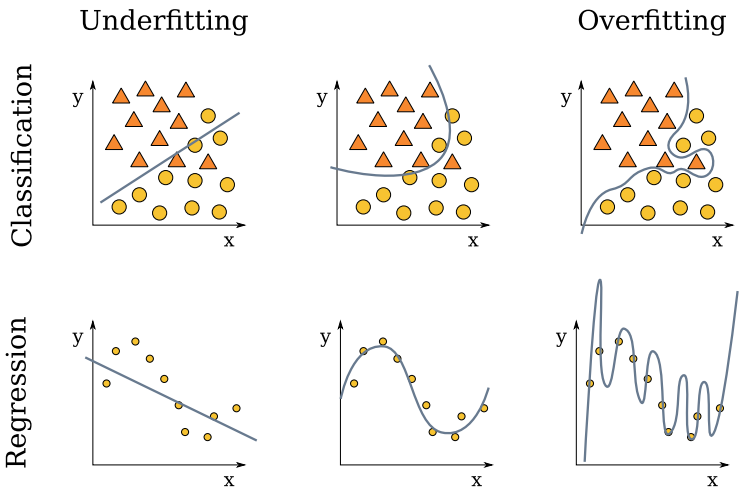

Figure 2.2.1: Overfitting and underfitting models. A model is said to underfit if it is not complex enough to capture the phenomena to be modeled and, at the opposite, it is said to overfit if it is so complex that learned the data without ability to generalize.

Overfitting and underfitting models often are unable to capture underlying phenomena

Overfitting functions are not often useful: they do not model correctly the problem and cannot be used to predict the behavior of the function $\hat{f}$ for a given input. The generalization abilities of a solution are very important when solving a problem that requires to sample new points from that solution, therefore evaluating the generalization performances of a solution is an important task. Naturally, one may evaluate those over new data if they can be acquired: this is what is usually done in a scientific context, where new evidence is used to disprove failing theses. If it is not possible to acquire new data, a possible evaluation process involves using two different partitions (i.e. with empty intersection) of the dataset: one partition is used to search for a solution, while another partition is used to evaluate its performances on unseen data. Getting rid of overfitting altogether is not possible in principle, but in this thesis we will cite few methods to reduce the chance of it happening.

2.3 Geometric Semantic Genetic Programming

When a solution f is evaluated on an input, it yields a point in its codomain, $\mathbb{R}$ in our case. Depending on the error measure we selected, that point will have a certain distance from the desired output, which is hopefully (but unlikely) zero.

The semantic of a function f is the values it assumes on the dataset $\mathcal{D}$

Assume now that the samples in the dataset have a fixed order, that is, each sample is numbered from 1 to m, being $m = \|\mathcal{D}\|$ the size of the dataset. Then, if we evaluate a solution over the entire dataset, we can build a vector

$$Y = [y_1, \cdots y_m] = [f(x_1), \cdots, f(x_m)]$$

which is called semantic. This m-dimensional vector can be positioned in the $\mathbb{R}^m$ space of vectors called semantic space, representing the space of points obtained by evaluating all the possible solutions over the dataset $\mathcal{D}$. The desired solution, $\hat{y}$ is, of course, a point in this space, and the distance between two semantics is the distance between two points of this space, under some metric.

The optimization problem over $\mathcal{D}$ can be state in terms of semantics

Adopting this view, the search for $\hat{f}$ is the search for a solution with a semantic that has zero-distance from $\hat{y}$. We can now define an equivalent optimization problem where $\mathcal{S}^S$ is the semantic space and $\hat{f}^{\mathcal{S}}$ is a function that measures the distance to the target semantic vector, $\hat{y}$. This function will assume non-negative values and the global optimum will be in zero, therefore it is a minimization problem.

Any non-optimal semantic has a neighbor which is closer to the optimum

Consider, in this case, the space of neighbors: if we consider an operator that mutates a solution by adding a random noise to any of its components, every solution but the global optimum will have a neighbor with a better fitness, because in any position of the space it is possible to identify a slightly different solution which is nearer to the best. We call this mutation box mutation, because on every axis of the semantic vector a noise can be added, ideally defining a "box" around a solution where neighbors are contained. A more rigorous definition of box mutation, with a mutation step μ, is the following:

$$\mathcal{M}(X) = X + U_m(-\mu, \mu)$$

where $U_m(-1, 1)$ is a vector of random numbers distributed in $[-1, 1]$ of length m.

This means that such fitness landscape becomes unimodal. As we explained earlier, this is a convenient situation and, although from a given solution it is not possible to determine a better neighbor without knowing the target semantic, this situation provides an enormous advantage in the search for a solution with respect to the original problem.

We look for operators that work on programs, but transform their semantic as desired

Naturally, solutions in this new problem are not programs: they are semantic vectors to which concepts of prediction and generalization do not apply. Semantics are obtained by evaluating a program over the entire dataset and we would like to find a program that is capable of producing such semantics because that program models the original $\hat{f}$ we are searching. Therefore we now have to find a way to map solutions from the original space $\mathcal{S}$ (programs) to the new space $\mathcal{S}^S$ (semantics) such that the genetic operators transform semantics as we described by acting on programs.

To perform such transformations on semantics, two semantic operators were introduced [72]: Geometric Semantic Mutation (from now on, GSM) and Geometric Semantic Crossover (from now on, GSC). These two operate on semantics by transforming the generating function by scaling and translating it.

GSM mutates a solution by adding a random noise to the function value

GSM, represented in Figure 2.3.1, operates on a function $f : \mathbb{R}^N \mapsto \mathbb{R}^m$ and, in the semantic space, has the effect of a box mutation and is defined as

$$f_M(X) = f(X) + \mu \cdot R$$

where μ is the mutation step, which is a multiplicative factor on how each component of the semantic will be scaled. Note that, in this case, μ does not impose a limit on the possible mutation of each component, therefore we are not actually bounding the box. R is a random vector centered in zero with length m, which effectively allows to apply a perturbation on each component of the original semantic vector. R is usually defined as a difference of two random trees $T_1 - T_2$ and sometimes a bounding function is applied to limit the value of its component in $[-1, 1]$, so that the mutation step μ is effectively the size of the "box" in which mutation happens. Good experimental results

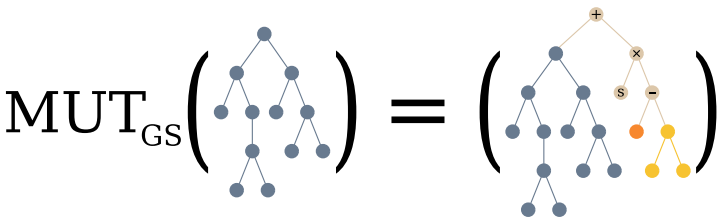

Figure 2.3.1: Geometric Semantic Mutation is represented. In this case, the difference of two random trees (in yellow and orange) is used to perturb the semantic of the individual being processed.

were obtained with this technique by applying a logistic map over the trees T_1, T_2.[14, 32]

GSC combines two solutions into one that has a semantic never worse than its parents

GSC, represented in Figure 2.3.2, works instead by interpolating the semantics of the parent individuals. Two parent individuals will have a semantic each that can be positioned in the semantic space as two points. It is possible to identify a single straight segment passing by those points, and all the points on that line can be identified via a single parameter a. Given two points p_1, p_2, all the points on the segment can be defined as

$$p = ap_1 + (1 - a)p_2$$

with $a \in [0, 1]$. This geometric interpretation can be applied to build the GSC operator, which combines two functions $f_1, f_2 : \mathbb{R}^N \mapsto \mathbb{R}^m$ using a random scalar value in the way we explained. This crossover has the property that the generated "child" individual will never have a worse fitness than the worse of the parents. A variant of this operator, used in [14] as well as in this work, uses a vector $\bar{a}$ with components bound in $[0, 1]$ therefore instead of being an interpolation of the two parents, the offspring will be a random point in the m-dimensional box defined by the parents. We have no experimental evidence that this variant is performing differently than the original formulation (theoretically speaking, this crossover is less effective, as it might introduce worse solutions in the population, but it could also produce a solution which is better than both parents, which the crossover as defined above cannot produce), and actually it seems that this operator is not useful to evolution [16], but in this work we built upon [14]. Therefore we maintain the same operators to

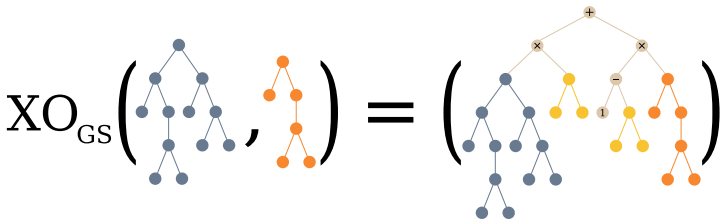

Figure 2.3.2: Geometric Semantic Crossover is represented as the weighted sum of the individuals being processed (in blue and orange); a random tree (in yellow) is used to weight each of the individuals to yield a semantic which is geometrically bounded in the sub-region of the semantics of the original trees.

exclude an additional factor that might influence our results, therefore making the evaluation of our algorithms more ambiguous.

GSGP is the GP algorithm that uses the Geometric Semantic Operators

Using these two operators, it is possible to produce a new generation of individuals that not only prevents degradation of performance, but will consistently improve the fitness thanks to the geometric properties of the semantic space. With Geometric Semantic Genetic Programming (from now on, GSGP) we will refer to a Standard GP with a fixed dataset $\mathcal{D}$ which maps individuals into a semantic space, computes the fitness as a distance function between individuals' semantic and the target semantic and uses Geometric Semantic Operators we described.

In GSGP solutions will grow without limits

Without the need for careful inspection, it is evident that these Geometric Semantic Operators combine solutions to improve their fitness, but the combination is not destructive. In other words, the size of each individual will grow very quickly due to the fact that each child encapsulates the full definition of its parents. In the field of GP, the tendency of solutions to grow in size unnecessarily is called bloating [4, 24], and often solutions are limited in size. In the case of GSGP, limiting size is not possible, but the quick growth of solutions make them hard to handle. Having solutions that continuously grow in size is not a new phenomena in GP: it is referred as bloating and it happens in Standard GP and other setups. Typically, some counter-measures are taken to avoid an undefined growth of the solutions, such as limiting the height of the solution trees so that neither crossover nor mutation will produce any individual with an height greater than the limit. This is naturally not possible

in GSGP but some techniques have been adopted, for example by applying algebraic simplification on the solutions [72].

Implementation of GSGP which can manage the growth have been implemented

The original implementation of GSGP could not be used for practical applications, but only for few generations on simple problems [72, 99]. This situation changed with the introduction of an efficient implementation [99], in which instead of combining full trees into new solutions, references (pointers) are used instead. In addition, to compute the fitness of a new individual, there is no need to store its representation at all: individuals can be characterized by their semantic, therefore space and time required by the evolution are $O(m \cdot N)$, being m the length of one semantic vector and N the number of individuals in the population.

The final solution of a GSGP run might be very large in size

The usage of pointers and efficient memory management might indeed make the evolution process of GSGP manageable and usable in practical applications, but even if the data store in memory are efficiently linked the reconstruction of the final solution will take a significant effort as it will require to unroll the entire tree. In addition, the amount of redundant or unused code might be very high, providing a little insight on the process and making the answer poorly intelligible. In this case, algebraic simplification of solutions is indeed a viable alternative.

2.4 On the Generalization Abilities of GSGP

Generalization is the ability to capture a relation among train data that holds also on unseen data

An important consideration must be done regarding the relation between GSGP and overfitting phenomena. Of course, geometric properties of semantics are valid independently from the dataset used as they are properties of metric spaces in general. In other words, independently from the dataset used, GSM and GSC can produce an offspring which is nearer to a given (optimum) point. Naturally, this works in a space where the optimum is known, and if we search for an approximated objective function in that space, it is not surprising that the approximation will perform well in such space. The question of generalization is if the same approximated function, which has the domain of the objective function $\mathbb{R}^n$, will be a good approximation on a different dataset for the same problem. For example, if we consider the entire $\mathcal{D}$, partition it in two subsets, then search for an approximated objective function on one of

the two partitions, obtaining a function with a certain error against the target semantic, will the same function yield a similar error when evaluated on the second partition against the second target semantic?

Experimental evidence shows that GSGP has good generalization abilities

Experimental results give an affirmative answer to this question and the explanation could be that the approximated function is actually able to capture the relations between input and output data. This is likely dependent on the semantic space used for optimizing the function: a 1-dimensional semantic space will very likely be insufficient to evolve an approximated function which is able to generalize, while a high dimensional semantic space will be likely more useful to approximate the objective function.

A low number of samples in the dataset will not capture the complexity of the objective

To better understand why this is so, and therefore understand whether GSGP is actually finding approximations able to generalize or not, consider a simple example problem, that will be used to provide a view on how GSGP works. In this problem the feature space (domain) is a uni-dimensional space, therefore $\hat{f} : \mathbb{R} \mapsto \mathbb{R}$ is the objective function we want to approximate. This function can be thought as a landscape, with hills and valleys, that will assume a value for all the possible inputs. A dataset is a set of points on this landscape: each point is characterized by its position x and by its elevation $y = \hat{f}(x)$. To start, consider a dataset with only 1 sample (one point on the map): its elevation in the landscape, the y coordinate, will characterize the global optimum point in the space of semantics. This space will be 1-dimensional, as we have only 1 sample of $\hat{f}$, therefore the optimum y value will lie on a straight line. If we were to find the approximate function $\tilde{\hat{f}}$ based only on this single point, any function that assumes that exact value on the same coordinates, that is $\tilde{\hat{f}}(x) = y$, would perfectly approximate the objective function.

Imagine now we sample 2 points on the landscape, each of them will have an elevation, which will represent a coordinate in a 2-dimensional semantic space. The global optimum point, in that space, is characterized by the elevation of the two points. If we were able to modify the landscape, raising or lowering some hills, we would see the points move up or down, and therefore the semantic point in the 2D space will change its position, on the plane, depending on how each coordinate was influenced by the shift of the hills. To have a better intuition of this mapping between the problem and the semantic

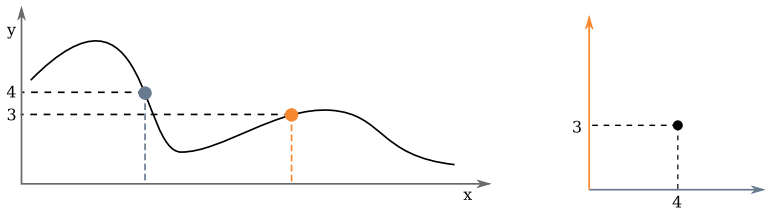

Figure 2.4.1: Mapping between the problem space (on the left) and the semantic space (on the right). Note how the dimension of the semantic space is defined by the size of the dataset (2, in this case), and not by the dimension of the problem input space (1, in this case).

space, consider Figure 2.4.1.

If we try to approximate $\hat{f}$ in these two cases, with 1 or 2 sample points, it will be rather easy to come up with an approximation that works on them, but it is very unlikely that the solution will generalize, as it means that our approximated function is a landscape that "looks like" the original landscape only in these few points.

With the increase in the number of samples, searching for an approximation means to search for a landscape that is more and more similar to the original landscape in the points we have sampled.

If we are actually able to fit a surface on the sample points obtaining a function similar to the original one, then it means that we have captured the relations between domain and codomain and the approximation can generalize on unseen points. This ability depends not only on the algorithm used to fit the points, but also on the number of points used (see Figure 2.4.2 for a visual intuition).

GSGP appears to effectively capture relations among data

The evolution process searches for better solutions by combining random solutions in a way that their performances are good, hopefully by exploiting relations between input and output data instead of mere chance. Experimental success of GSGP, including its remarkable ability to generalize, encourage to think that the individuals selected by the evolution process, obtained by the means of GSM and GSC, are in fact exploiting such relations in data, and not produced only by random guessing. As with most Machine Learning models, GSGP does suffer from overfitting especially in later generations, therefore it is possible, if not likely, that the evolution process exploits both casual

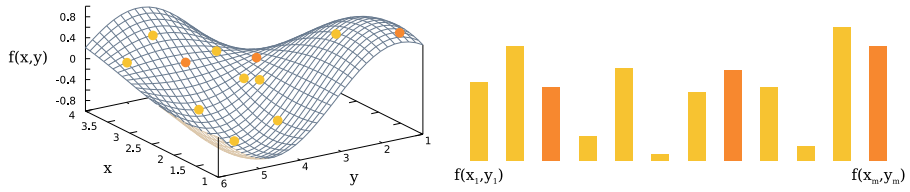

Figure 2.4.2: A representation of the semantic of a function $f(x, y)$. The function is usually unknown (thus, it is the objective), but a number of samples are given, therefore we know the values assumed by the function for a given set of input values, $(x_1, y_1), (x_2, y_2), \ldots, (x_m, y_m)$. The set of known values is the semantic of f. The number of samples is crucial to being able to capture, and therefore to approximate, the unknown function (compare orange and yellow samples).

and non-casual relations in data. A possible explanation for the generalization abilities of GSGP is that it exploits the information in the dataset by acting as an ensemble: GSM and GSC are performing a combination of the data, preserving the information and while mutation operates on a single data point, crossover combines the information from more points. This hypothesis was also suggested in [32].

There are strategies to reduce the risk of overfitting in GSGP

To further lower the risk of overfitting during the evolution, some strategies can be applied, ranging from using different subsets of the original dataset [34], to the use of a bounded mutation step in GSM [32]. Normally, it is very unlikely that an individual will overfit since the beginning of the evolution: in Standard GP and GSGP, individuals will be randomly generated and their performance will usually increase with time, and overfitting appears as a a consequence of the evolution of overly-complex solutions. In this thesis we will deal with a different case, in which it is likely that individuals will be overfitting since the initialization and we will propose a strategy to counteract this phenomena.

2.5 Ensemble, Meta-learning and Hyper-heuristics Methods

For later discussion, it is also useful to briefly review what Ensemble, Meta-learning and Hyper-heuristic methods are, how they are characterized and what are the differences among them.

An Ensemble Method (from now on, EM) is generally described as Machine Learning method in which multiple models, often referred as Basic Learner (from now on, BL), are trained and combined in some way to yield a model which generally works better than any of them, taken singularly. Key aspects of EMs are the presence of multiple Machine Learning models and the presence of a higher level model which is capable of combining them and producing meaningful results. Ensembles are often effective, producing better results than single models in many practical applications, see for example [8, 9, 105, 107, 108].

To increase the effectiveness of ML models, they can be ensembled in many different ways

There are of course many variations of EMs and multiple classifications have been proposed. We consider here stem from the analysis and reviews by [48, 85, 87, 92, 107]. The first classification method we consider regards the representation of the input: EMs can use the same or different representations for the input. A theoretical analysis of ensemble classifiers based on same or different representations is presented in [43]. When using the same representation for the input, different classifiers try to build a different estimate of the same *a posteriori* probability distribution, while if different representations are employed, especially if they represent different features or physical measurements, this view cannot be adopted and different rules can be used to combine the outcomes of each BL. Another possible classification is whether the EM being considered is an actual ensemble or if it is a modular system: in the latter, a problem is decomposed into sub-problems which are solved by different modules of the system. Alternatively, in [85] proposes another classification regards that considers diversity is obtained in BLs: it could be an implicit feature of the method, but it can also be obtained explicitly by a process which involves a measure of diversity. In addition, in the same work, EMs are also classified depending on how they combine BLs: if an additional training effort is required to combine them, then such EMs are called trainable, otherwise if they can combine BLs as they are introduced, without requirements of an additional tuning, the methods are called non-trainable. Another possible classification of EMs, proposed by [107] is whether BLs are homogeneous or heterogeneous: in the former case, the method uses models of different nature. For example, Random Forests is an EM that uses multiple Decision Trees to obtain better performances. In this case, the method is

homogeneous, because the BLs are all of the same nature: decision trees.

Meta Learning focuses on using ML to tune and combine other ML models

Similar to EMs are the Meta Learning Methods (from now on, MLM) and Hyper-Heuristic Methods (from now on, HHM), which were developed independently for some time, but are similar in many ways and, lately, are heavily influenced one by the other [77]. In MLMs, the challenge is learning to learn: usually, this is done by using Machine Learning over the metadata of other Machine Learning algorithms. For example, by applying an algorithm that identifies which parameters, operators and functionals shall be adopted by GP to solve a certain class of problems. A similar approach is being used in HHMs, in which the objectives and methods have much in common with MLMs, but the focus was historically on optimization problems, using heuristics instead of machine learning as a basic unit to solve problems. With time and progress, GP assumed a fundamental role in both of these fields, and they are currently sharing many traits. Generally speaking, the main differences between MLMs and HHMs are the field of application (MLMs are often focused on classification problems, while HHMs are mostly focused on optimization problems), the kind of representation used and the basic units of problem-solving (Machine learning models on one side, heuristics on the other). For example, some of their differences stem from the fact that optimization problems and classification problems have a different nature, and while we can establish some evaluation measures that are shared by all classification problems, it is not possible to do the same with optimization problems. In spite of this, both of them share the objective to develop more general algorithms, which are capable of solving different instances of the same problem or, even better, different classes of problems.

Models can be generated, selected or combined according to a learning process

As highlighted in [10, 77], we can typically find two subsets of MLMs and HHMs: methods based on a selection of their components, or based on the generation of their components. The former are methods that will use pre-existing learners (or heuristics) and select the one most fit to solve a given problem. On the other hand, methods based on generation will build learners or heuristics to solve a given problem. There are also methods based on a combination of their components, for example consider Boosting [91], Bagging [8, 9] or Stacking [105]. These methods are widely used and exploit a combination of their components to take decisions: for example, bagging uses

a majority voting to determine the class a given sample belongs to, based on the votes of each of its basic learners.

Thanks to its characteristics and versatility, GP and its derivatives are appealing techniques for MLM and HHM applications and they have been used extensively [28, 47, 62, 65, 65, 78, 81, 83, 101, 106].

3

Universal Genetic Programming

The main contribution of this thesis stems from the following question:

> semantics are vectors that can be generated from any valid program; what happens if that program is another Machine Learning method, previously run over the same dataset?

Using additional ML methods to initialize individuals turns out to be a good choice. We will now explain the reasons, advantages and drawbacks of doing so and in the next chapter we will discuss experimental results sustaining our thesis.

3.1 Initializing GSGP with external models

UGP uses the output of pre-trained models in its initial population

Universal Genetic Programming (from now on, UGP) is a simple extension to GSGP in which some individuals are not initialized randomly, but are initialized by means of external programs. Given a set of ML models $\mathcal{M}$, they can be trained over the same dataset of GSGP to automatically learn correlations in data, to predict output values from inputs. The models are trained

over a training dataset, which is completely separated (empty intersection) from the testing dataset, which can be used to evaluate the performances of a given model. These two datasets are partitions of the entire data we have on a problem, the dataset $\mathcal{D}$, and can be used by any suitable ML algorithm, including GSGP.

Seeding is the usage of pre-made individuals in initial population

The idea of combining different ML models is, as we saw earlier, key in the fields of Ensemble Learning and Meta-Learning, that deal with defining models that can be combined or tuned to yield generally superior performance to every single model. More specifically, the technique of using pre-computed individuals is not new also in the GP community [1, 38, 51] and it is often referred to as *seeding*, although it is the first time it gets applied in the semantic context, with a deeply different meaning and significantly different results.

Initialization uses dummy individuals whose semantics comes from external programs

Going in detail, UGP is GSGP with a single variation in the initialization operator: in this case, the operator will not generate all individuals randomly, but will initialize the first $\mathcal{M}$ individuals using dummies whose semantic is obtained by evaluating the models in $\mathcal{M}$ over the training dataset. This is possible because, as we illustrated earlier, GSGP does not require to store the entire tree for one individual, but only a limited number of semantics is necessary at any given time.

Individuals that overfit since the initialization will compromise the entire evolution

As we illustrated earlier, GSGP is well-behaved regarding overfitting, but still suffers for it. There are techniques to make GSGP more robust against overfitting, but there is one case in which similar techniques are ineffective. This case, which very rarely occurs in Standard GP, is when an individual is overfitting early in the evolution process. In such case, the fitness of that individual will be very high if compared to other individuals, but its performance on the testing dataset will be bad, and likely it will worsen with time.

Selection operator is unable to discriminate overfitting individuals

In these cases, the overfitting individual will be highly preferred over the others in the evolution process, especially if elitism is enabled which will likely guarantee its survival from one generation to the next. With a high chance of survival comes a high rate of reproduction, which will ultimately skew the population toward that solution, very likely without improvements in the quality of the solution. Let us remark how the Selection Operator is not sufficient to deal with this problem, because it uses the training dataset to evaluate individuals, therefore introducing a bias that prefers individuals which are per-

forming well on that dataset and therefore might overfit when evaluated on a different dataset.

Different datasets are required to train and validate models

Since overfitting can be measured only against a test dataset, which must not influence the evolutionary process, it is in principle impossible to both train individuals and prevent overfitting with a single training dataset. What can be done is to estimate how much individuals are overfitting by using an additional dataset: instead of using train and test datasets, the $\mathcal{D}$ is partitioned into 3 datasets which are used for train, validation, and test. Train datasets are used to train models over data, while validation are reserved for some kind of model selection (such as identifying overfitting individuals). Test datasets are used to evaluate performances without any impact on how the ML model is obtained. Being initialized by an external procedure, it may be possible that a model overfits even before the evolution starts (this actually happened in our experimental settings). Therefore a possible strategy, which was adopted in this work, is to use the validation dataset to evaluate the performances of BLs after being trained on the training dataset.

Once models have been evaluated, they must be selected to avoid overfitting ones

Once individuals have been evaluated over the train and validation datasets, it is necessary to determine which individuals will be included in the evolution. Again, different strategies can be used. One simple strategy is to discard individuals if they are under-performing on the validation dataset: if training performances are good, but validation performances are not, it is likely that the algorithm overfitted and is unable to generalize. In this case, overfitting individuals can be discarded and will not be included in the initial population. This approach scales well with the number of models, and if many models are used it is even possible to compare their performances to determine which ones can be discarded (e.g. if validation performances are too far from the average). A different strategy, which is computationally expensive and does not scale well with the number of models, but it is statistically more accurate and gives some useful insights, is to evaluate the performances of every possible combination of models after a brief run of GSGP.

Note how this process is similar to how many ML practitioners select their model for a given problem: they identify the candidates, which are then trained over some training data, then evaluate their performances over unseen data to identify if they are overfitting or if they are able to generalize. We are substan-

tially automating this process.

We will now describe the strategy adopted in this work, to which we will refer with Pre-Evolution Selection Process (from now on, PESP), which consists in selecting a combination of models in which each model was individually having good generalization performances after training and after a short run of GSGP. We refer to UGP adopting this selection strategy as Universal Genetic Programming with Selection (from now on, UGPS).

3.1.1 Parameters and notation

To better describe how UGPS works, its parameters and notation are introduced. For a compact description, please look at Table 3.1.1.

When performing an experiment E using Universal Genetic Programming with Selection, it is likely that the algorithm, being stochastic, will be run multiple times over the same data, with the same settings. We denote the number of runs of the algorithm with R, and r_i is the i-th run of the algorithm.

GSGP is used both when selecting overfitting individuals and when evolving an ensemble

UGPS will use the GSGP algorithm in two phases: a selection phase, in which PESP will occur, and a "long" phase, in which GSGP runs with the objective of improving the solutions. In each phase, the number of generations used will be different, and we will denote G_s the number of generations used in the selection phase and G_l the number of generations used in the long phase. It holds that $G_s \leq G_l$ in every scenario. We will refer to each BL as $m_i \in \mathcal{M}$, being $\mathcal{M}$ the set of supervised ML models to be used as Basic Learners. The entire labeled dataset available in the experiment E is noted as $\mathcal{D}$, as earlier. This dataset is a single entity, not yet partitioned into training, validation or test datasets.

A nested K-fold cross-validation is used to partition the dataset for training, validating and testing

In this work, we will adopt a K-fold cross-validation in a nested fashion to partition the dataset. The reason for this choice is exclusively to perform robust estimation of the behavior of the algorithm at the expense of computational effort. This strategy is preferred over a simple three-way split of the datasets also due to the limited size of some datasets. In this way, we will use maximize the amount of data used to train and validate each BL, but without sacrificing the data used to train and test their ensemble.

Nested K-fold allow us to use more data for training an ensemble than a 3-way partition

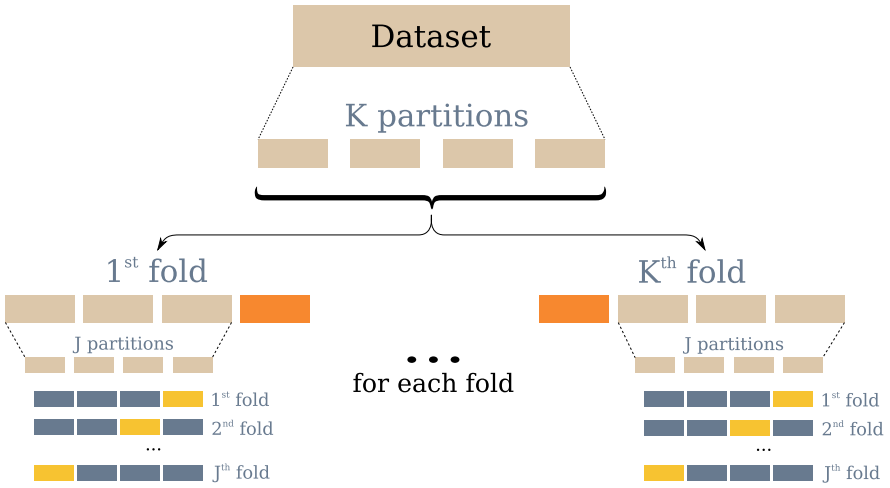

Figure 3.1.1: To train the models and then select them, the data should be split in 3 partitions: training, validation and testing. We use nested K-fold cross validation. In this representation, data that is used for training is represented in blue, data that is used to validate the models is in yellow and test data to evaluate the overall performances is colored in orange. Each block is a partition p of the data and folds are union of different partitions.

This being said, it is important to highlight that the partitioning scheme we are adopting to train the models, validate and test the algorithm to estimate its performances are *not* required by UGPS, and a different scheme can be used. An alternative which is typically used when training, validating and testing a model is to perform a random split of the dataset, partitioning it in 3 subsets, dedicating, for instance, 60% of the dataset to the training of the models, 20% to the validation and 20% to the testing that is, estimation of the generalization abilities of the algorithms. This process does not however guarantee that all data is used for both training and testing and it also reduces the amount of data available for training.

Each of the training K folds is J-folded yielding a total of $K \times J$ partitions

The dataset $\mathcal{D}$ will be partitioned in 2 partitions in K randomly selected ways (folds). Each fold will be denoted with d_i and it is a partition of $\mathcal{D}$ comprising two subsets of $\mathcal{D}$: d_i^t, the i-th training partition and d_i^v, the i-th testing partition. Since the partitioning of the dataset is nested, each training partition will be J-folded and we will denote with d_{ij} the j-th fold of d_i. Similarly, as we defined above, we will denote with d_{ij}^t the training partition and with d_{ij}^v the validation. For a visual representation of how data is partitioned, refer to

Figure 3.1.1.

Symbol	Meaning
E	experiment being performed
R	number of runs in E, r_i being the i-th run
G_s	number of generations (short/selection)
G_l	number of generations (long)
M	set of m_i machine learning models to use
$\mathcal{D}$	entire dataset available for experiment
K	number of cross validation folds, d_i, over $\mathcal{D}$
d_i^t	i-th fold of $\mathcal{D}$, training data partition
d_i^v	i-th fold of $\mathcal{D}$, testing data partition
J	number of nested cross-validation folds over d_i
d_{ij}^t	j-th fold of d_i^t, training data partition
d_{ij}^v	j-th fold of d_i^t, validation data partition

Table 3.1.1: The notation used for describing the partitioning scheme used on the datasets.

Selection uses the nested J-folds, while evolutionary ensemble uses K-folds

The PESP, starts by partitioning $\mathcal{D}$ in K subsets p_i which will be combined into a training dataset $d_i^t = \bigcup_{k!=i} p_i$ and a testing dataset $d_i^v = p_i$. Each pair of d_i^t, d_i^v is referred as "fold". In this way, each instance of the dataset will be used exactly $K - 1$ times for training and 1 time for testing. Each training partition d_i^t is then partitioned again, in the same way, using a similar scheme, with J partitions instead of K, yielding J folds d_{ij}^t, d_{ij}^v. The selection algorithm then proceeds by training the ML models over the nested training partition d_{ij}^t and, when done, models must be evaluated on validation data d_{ij}^v to reduce the chances that overfitting ones are included in the long phase. Once selection is concluded, the evolution will use the entire training data of each fold d_i^t to train the models in the long phase, and the overall performances will be evaluated on never-seen data d_i^v.

So, consider for example the case in which we are evaluating all combination of models. If we have the set of models $\mathcal{M} = \text{LR}, \text{MPL}, \text{SVR}$, each of them will be trained over d_{ij}^t and then used in some combination of models. In this case, we will have $2^3 = 8$ combinations. Each combination is used with each partition d_{ij}^t to run UGP. For instance if we have a 5-fold nested cross validation, each $M \in \mathbb{P}(\mathcal{M})$ will be used for each fold, for a total of

$5 \times 8 = 40$ runs of UGP. This means that for 5 times, UGP will run with no models in its initial population (corresponding to $\emptyset \in \mathbb{P}(\mathcal{M})$), each run using $d^t_{i,0}, \ldots, d^t_{i,4}$. Then the set of models $\{\text{LR}\} \in \mathbb{P}(\mathcal{M})$ is used 5 times in the initial population of UGP, each using the folds $d^t_{i,0}, \ldots, d^t_{i,4}$ again. This process continues until all the subset of $\mathbb{P}(\mathcal{M})$ have been used, for a total of 40 runs. Once a combination of models is used for all the folds, we can evaluate the impact of the models on the evolution by observing how they behaved on the validation datasets d^v_{ij}. When all the 40 runs end, we can average the results of the models on the validation datasets, yielding an average performance for each of the 8 combinations of models. We can now see which of the combinations is the "most promising": overfitting models will have bad performances, while models which are synergistic and produce best results will have high performances. Given this knowledge, the most promising combination of models is used for a long run of UGP. This time, since we know what is the best combination of models, we will use the entire (non-nested) training dataset, d^t_i, for G_l generations. The performances of this long run are then evaluated over the testing partition d^v_i to estimate how the model behaves over unseen data. Note that the validation partition, d^v_i was never used up to this point. This process, composed by selection and long run, is then repeated for each of the K folds, so that we will have good statistics on how the models behave. Since the folds are randomly selected, to rule out the chance of lucky partitions, the entire process can be run R times. This concludes the experiments E with a small amount of bias.

Evaluating each combination is expensive, but gives hints on synergic behavior of models

This selection strategy, obviously, does not scale well with the number of models. In the selection phase UGP runs, even for G_s generations, a total of $2^{|\mathcal{M}|} \times J \times K \times R$ times, which is prohibitive for real-life applications. In addition, it requires a certain number of parameters which must be selected and tuned. Nevertheless, this approach could be interesting in our case because it gives us some additional insights: not only it allows to evaluate accurately UGP performances when compared to GSGP, but it also highlights how problems and models relate. We can anticipate, for example, that models may have synergistic behavior: using a combination of models often outperforms the solutions obtained by the same models taken individually. Also, in all the datasets under observation, this selection procedure highlights how a

combination of models is always preferable to no models (i.e. GSGP), suggesting that UGPS is indeed an improvement over GSGP (and UGP, since it lacks overfitting control).

Each model will be evaluated alone and the best half of models will be included in the long run

Since evaluating a large number of models is a priority in this work, we cannot possibly explore the $2^{|\mathcal{M}|}$ combinations of models. Therefore, for the evaluation of GSGP with an higher number of models, we adopted a different strategy which is much simpler: every model is tested alone, without evaluating any combination of them. In this way, the complexity scales linearly, being $O(|\mathcal{M}|)$ instead of $O(2^{|\mathcal{M}|})$, although we cannot evaluate synergistic behaviors. Once each model has been evaluated, it is necessary to pick one or more of them to be used in the long run. In this case, we selected the top 50%-best performing models and excluded the other half. This should reduce significantly the risk of picking overfitting models, especially if the models included are very heterogeneous, as in our case. This might not be the best strategy if the models were more homogeneous, for example if many of them were of the same nature, but with different parameters. In this case, one might select a lower number of models (for example, the top-3, top-5% or just the best one).

Other strategies that could be adopted, but we did not test, could involve random sampling combination of models, incremental grouping of models or tournament-based selections.

3.2 UGP as an Ensemble or Meta-Learning Method

UGP and UGPS can be classified as EM or MLM

UGP and UGPS can be seen as some sort of EM, as their founding principle is to use, in addition to randomly generated solution trees, additional algorithms which are combined into better solutions. On the other hand, the use of GP is diffused in Meta-learning and Hyper-heuristic methods, therefore it is also of interest to analyze how UGP and UGPS can be classified with respect to the current state of the art of these fields. In this section, we will try to classify our algorithms according to some criteria normally used for EMs, as well as verify if they can be described in term of Meta-learning or Hyper-heuristic methods. There are different categories into which they could be classified, but it is not always easy to agree with the classification and, instead of trying to apply a given taxonomy to UGP and UGPS among the ones present in lit-

erature, we will consider a number of classification criteria independently.

The proposed algorithms are hybrid ensemble methods

Strictly speaking, both UGP and UGPS use the same representation for the input, using all the available features. Interestingly, using semantics as an intermediate representation, we could argue that on two different levels the representation used is of the same kind. Nevertheless, we cannot guarantee that BLs adopted will adopt the same representation given the same input. Although we do not exclude features *a priori*, it is possible that different BLs will try to approximate different probability distributions by adopting different sets of features. For example, we could design two BLs of the same nature (e.g. linear regression), but using two different subsets of input features. Therefore, according to the EM classification proposed in [43], we argue that UGP and UGPS are hybrid methods that might be considered to use the same input representation or not, depending on the adopted BLs.

UGP and UGPS do not directly rely on the structure of a problem and its subproblems

If we consider classification by modularity, it is clear that both UGP and UGPS are ensemble, and not modular systems, as each BL ideally solves the same problem and their results are combined to yield a final solution. There is no partitioning of the problem to be solved and the system is trying to solve it globally, at once. When we consider how diversity is obtained, it is again possible to identify UGP and UGPS as methods that achieve the diversity by an implicit mechanism, being diversity part of the evolutive process and therefore a fundamental characteristic of any GP system. Nevertheless, different techniques have been proposed to explicitly increase diversity in the population during the evolutive process and, if these methods were to be adopted, our classification could be arguable.

They can also be classified as homogeneous and trainable EMs

If we were to classify the proposed algorithms in terms of how BLs are combined, UGP would be classified as a trainable EM as it is the evolutionary process itself that tunes the combination of models so that they are effective problem solvers. It is interesting to note how the combination of BLs is very deep and rooted in the semantic representation: after the evolutionary process, the learners lose their individuality and are transformed into larger, combined entities. UGP does not perform a simple weighted combination of models: the evolutionary process improves on the models, possibly combining them into radically different entities and the combination of models might as well be as complex as the models themselves. In this, we found that UGP is rather

unique and we are not aware of other ensemble method with this peculiarity. When it comes to the homogeneity of EMs, it is obvious that UGP is heterogeneous, unless the BLs are obtained by running UGP or GSGP via seeding or restarting.

Based on the classification of Sharkey [92], an interesting classification for UGP and UGPS is whether the EM is built on the direct combination of BLs outputs. In this case, we think that by using semantics, our algorithms can be classified as EMs that directly combine the outputs of the BLs and do not work, for example, on how the BLs operate. The learners are black-boxes and UGP (UGPS) works exclusively with their outputs. To conclude the possible classifications for UGP and UGPS, we would like to consider whether the EM are generative or non-generative, a classification which is related to the previously mentioned one, by Sharkey. In the former category, we can place EMs that generate different BLs using their respective learning algorithms. On the other hand, non-generative models do not try to generate BLs to maximize their performances, but combine them, for example by majority voting, and depend on their performances. We think that UGP and UGPS fall in the non-generative category, because both of them do not try to generate new models.

Our algorithms do not rely on internal representation of BLs, they combine their outputs

We can therefore state that UGP is a trainable, heterogeneous, non-generative EM that combine the output of their BLs, while the distinction on other criteria is not completely clear.

As we saw in Section 2.5, GP is widely adopted in MLMs and HHMs. This is possible thanks to the flexibility of GP in working with any representation for a given problem. For GSGP, UGP and UGPS the situation is a bit different, because they work directly with semantic vectors and the representation of solutions is partially enforced. This limits the role of UGP and UGPS in their role to build Meta-Learning Hyper-Heuristics, but allows them to assume a new role as we shown in this work. We can indeed refer to UGP and UGPS as a meta-learning method in the sense that it is a ML method that has the objective to combine and tune other ML models. As we will see in the next chapter, it is also effective in doing so.

3.3 Implementation details

Our implementation is based on an optimized version of GSGP

It is worth to briefly discuss also some implementation details of UGP and UGPS, as the contribution of this work is not only in terms of developing a new EM, but also in developing a tool that can be fast and useful to GSGP practitioners. The algorithms are based on the code of GSGP in [14][1], but our implementations are a complete rewrite in Go programming language. In general, our code improves the older one in maintainability, correctness, features and user experience: the core algorithm has been implemented in sensibly shorter and cleaner manner, some small bugs were corrected, with impacts on both computational performances and algorithm precision, making our GSGP version producing better results in some initialization cases. In particular, we corrected a bug that caused the best individual to be added twice in the population and an out-of-bound bug that caused incorrect memory access, both of which might have slightly impacted the outcome of the algorithm, but we have no quantitative analysis to determine what is the numerical difference.

The implementation is backward-compatible, but extends and improves the usability

The tools exposes a standard Unix-like Command Line Interface (CLI), by introducing flags as well as improving the previously used configuration file system, which can be used in combination with flags. In addition, two Python scripts were introduced, one to launch experiments using UGPS and a sibling script that collects and analyze results, which are made available in a Python Pickle file. All the plots and results available in this thesis were produced using these three tools[2].

Go proven to be a suitable language in term of speed and maintainability

Go was chosen for several reasons: first, it includes a semi-automatic memory management system, which excludes dangling pointers and garbage-collects unused resources; it imposes strong typing conventions as well as software engineering good practices which help to keep the code clean and maintainable. In addition, it contains a compiling sub-system, CGO, which allows easy interoperability between Go and C, even if their calling conventions are different. This aspect was indeed useful when writing a version of UGP which uses NVIDIA CUDA for massive parallelization of vector operations, although we

[1] The code is available on-line at http://gsgp.sourceforge.net/

[2] Our implementation of GSGP, UGP and UGPS are freely accessible on-line at https://github.com/akiross/go-gsgp

used the gccgo compiler in this case, which proved more robust in this context. Go also includes facilities to write correct and performing concurrent programs, via an abstraction called goroutines, which behave like extremely light threads and can span multiple cores on a single CPU. Performances in Go are very good, even with a garbage collector, but accurate memory management can reduce its workload by providing additional speed. In fact, Go was shown to be one of the best languages, performance-wise, for evolutionary computation in [66].

We improved computational speed of existing solutions with extensive parallelization

We used single allocations, pointer swapping and temporary buffers to reduce the garbage collection to a minimum. Even without employing multiple cores and with computation of additional statistics (such as the computation of contribution, which uses arbitrarily large integers), our implementation is 28% faster than the original GSGP implementation referenced in [14].

Our imlementation exploits the inherent parallel nature of GP algorithms

By enabling the parallelism, the performances can be even higher and it scales with the number of available cores. GSGP can be parallelized in different ways. For starter, individuals can reproduce, recombine and mutate in parallel, as individuals in the next generation depend only on individuals in the current generation and only read-access is needed, making these operators thread-safe on many currently used architectures. Note that many Evolutionary Algorithms are inherently parallel by nature, and GP is no exception. In fact, many different highly parallel implementations exist, both using CPUs and GPUs, in addition to distributed systems [35, 50, 74, 101, 104]. Another aspect which can be parallelized is the fitness evaluation: each individual can be evaluated on the entire input in parallel, presenting each input of the dataset to each individual and collecting the result. The only important aspect here is to maintain a constant mapping between the order of input samples in the dataset and their evaluated results in the semantics. This aspect, fitness evaluation, is often the prominent effort computationally-wise: it is usually the most compute intensive part of the evolution and it is the one we chose to parallelize.

Parallel fitness evaluation provided significant speedup with

Our implementations of GSGP, UGP and UGPS are parallel in the evaluation of fitness and have been parallelized both using CPU, via concurrent design with goroutines, and using GPU via CUDA; in both cases, each component of the semantic vectors is computed in parallel. We adopted this strat-

egy for several reasons. First, datasets are getting bigger and bigger as time passes, therefore it is more probable that in the future there will be a necessity to scale over the dataset size instead than having a larger number of individuals. Related to this is the fact that usually the semantic is much larger than the number of individuals and, even if it is not, operators on individuals are applied $O(N)$ times each generation, with N being the number of individuals, while evaluation of individuals are computed $O(N \times M)$ times, with M being the size of semantic. Also, even if the datasets are growing in size, so are memories available on GPUs: in our cases, all the datasets fits in GPU global memory and therefore can be copied once over the computation device and be used for evaluation with minimum overhead.

Parallel fitness evaluation has been implemented both in CPU and GPU

CPU implementation has been done by simply performing in parallel the evaluation of each input sample for each individual: a number of "workers" goroutines are created, by default one for each CPU core available, and each of them will evaluate one individual on a share of the dataset. Individuals are evaluated in sequence. GPU implementation is different in the sense it loads all the data into the device memory and all the semantic computation is performed on the device. In fact, the evaluation of individuals just requires the fitness value and how it is computed is not relevant to the ends of the evolutionary process. This means that we can actually place all the semantic related code onto the GPU, perform semantic evaluation on it, and then just return the fitness value of each individual onto the host. This approach yields a high usage of the GPU and minimizes data transfers between host and device, an operation which is usually slow as it is limited by the bus capacity, which is orders of magnitude slower than memory transfers.

3.3.1 Reproducibility of Results

We used solutions such as Docker and Open Source to ensure reproducibility of our results

The implementation work was focused on building correct software and test it under reproducible conditions. For this reason, in addition to the algorithm, tools were developed to run tests and perform analysis automatically. We used Docker to build an image with all the software, its source code, data and configuration used to run all the tests. Any result presented in this thesis is completely reproducible by running a single Docker image[3].

[3] The Docker image used is available at https://hub.docker.com/r/akiross/ugps/

Transparency and openness are a minimum standard to achieve reproducible science

Of course, there might be different opinions over what ”reproducible science” is, consider for example the discussion by Drummond [26] in which reproducibility is considered a higher objective than replicability, being the latter a mere reproduction of existing results, with no change and no progress, as instead it happens in the former. The semantic associated to these terms may vary, for example in [79] it is argued that the “attainable minimum standard is ”reproducibility”, which calls for data sets and software to be made available for verifying published findings and conducting alternative analyses”, which contrasts with Drummond’s interpretation. Nevertheless, we agree that providing all the required elements to produce the same results that were published is fundamental to enable verification of one’s work and scientific progress based upon it. For this reason, the Docker images used to produce the results of this thesis are freely available and inspectable, including code and data required to produce the results.

4

Experimental Results

In this chapter, we will present the experimental results obtained running GSGP, UGP and UGPS over five different datasets. We will introduce those datasets, illustrate the experimental settings, the results of running the algorithms and we will discuss them.

4.1 Datasets

Oral Bioavailability (from now on, bioav) The first dataset considered is related to an important characteristic of drugs: their oral bioavailability. Differently to drugs injected directly in the blood stream, orally administered drugs have multiple obstacles to overcome before reaching systemic circulation. The fraction of oral administered drug that makes it, is the oral bioavailability and is denoted with F% (or %F). Increasing this fraction will lead to more efficient usage of the drug, lowering the required dosage, lowering the risks of possible side effects, reducing toxicity as well as inter-individual variability (i.e. the variation of effects of a drug among subject individuals). Oral bioavailabil-

ity is a fundamental pharmacokinetic parameter of a drug when it is designed and developed. The problem of predicting F% has already been faced in the GP community [2, 3, 14, 99], therefore we will compare our results to the ones obtained with GSGP in literature.

High-Performance Concrete (from now on, concrete) Concrete is, in general, a composite material which different mechanical properties that depend by its composition. High-performance concrete, in particular, includes additives which make the formulation even more complex. Modeling its mechanical properties starting from the mix is a hard problem, as it depends not only on characteristics such as the water-to-cement ratio but also on the presence and the quantity of other cementitious materials. Requirements for specific applications, regulations, costs and availability of materials are just some of the constraints that must taken into account when evaluating a specific high-performance concrete for a practical application, and the lengthy process of putting various concrete compositions under mechanical test can be expensive and an accurate model is desired. A recent publication[12] used GSGP to model and predict the strength of high-performance concretes using machine learning. We will use their results as benchmarks and show how our algorithms can improve the results.

Energy Performance of Buildings (from now on, energy) A large fraction of the global energy consumption is due to thermal regulation in buildings, especially HVAC (Heating, Ventilation and Air Conditioning). Heating Load and Cooling Load are two important indicators of the energetic effort required to maintain a suitable temperature inside a building, and they are kept into account by engineers and architects when designing a new building. Often, simulations are used to estimate energy consumption of a building keeping into consideration all the required variables such as occupancy, size, climate, humidity and of course the characteristics of the building being designed. Although this approach allows to study and model conditions that would be hard to have in practice, and is also accurate in comparison with real cases, it can be very time-consuming. For this reason, ML approaches have been used to model the energy performance of buildings and to predict consumption such as Heating Load and Cooling Load. The dataset we are using has been built synthetically using reliable simulation programs, and takes into account

parameters commonly used by designers to estimate energy performances. More details about the dataset can be found in [96], while reference results for GSGP applied to this dataset can be found in [15], to which our results will be compared.

Parkinson (Motor) (from now on, pkm) and Parkinson (Total) (from now on, pkt) One of the most common neurodegenerative disorder is the Parkinson's Disease, which leads to severe impairments to body motions. With the increase of average population age, the number of people affected by this disease is expected to increase and so are the costs of medical care. It is currently not possible to treat the disease, but the therapies available can improve the functional capabilities of a patient. In this process, keeping track of the symptoms is fundamental and the most commonly used system to measure the severity of the disease is the Unified Parkinson's Disease Rating Scale, or UPDRS, which gives a unified view on the severity of different symptoms in a scale from 0 (healthy state) to 176 (total disability). UPDRS is divided into three sections: a) mentation, behavior and mood, b) activities of daily living and c) motor. Estimating the severity of symptoms of a patient is a long process which requires expensive specialized examinations, and therefore automated estimation techniques would be preferable. In this work, we will apply our algorithms to the prediction of UPDRS scores for the motor subsection as well as an overall index and we will compare our results with previous scores obtained via GSGP [13].

4.2 Experimental settings

We show the improvements over the state of the art by comparing UGP and UGPS with GSGP

On each dataset, we run both the algorithms we are proposing, UGP and UGPS, as well as their funding algorithm GSGP which will be used as a reference. The parameters related to the evolutionary process, such as the size of the population or the mutation rate, can vary among each dataset used; those parameters often have a significant impact on the performances (see, for example, [18, 39]), therefore where possible we used the same values used in literature. A description of the involved parameters is present in Table 3.1.1, while some instances of their values are presented in detail in Table 4.2.1 and will be discussed later. Please note that the fitness function used was the error

measure between the output predicted by the algorithm and the target value of the dataset. The function used was either Root Mean Squared Error (RMSE) or Mean Absolute Error (MAE), depending on the error function used in the reference literature publications, to which our results are compared.

60 runs for each experiment, using 5-4 nested fold, with 11 standard ML models

All the tests have been executed using a 5-fold cross-validation ($K = 5$) and, for UGPS, a 4-fold nested cross-validation ($J = 4$); these number of folds are not commonly used, but this choice can be explained by the size of the datasets, which are not particularly large, and the necessity to partition each fold into nested sub-folds, making each d_{ij}^t of size $\frac{1}{5}\frac{1}{4}$ of the entire dataset. Each experiment included $R = 60$ runs, to increase statistical confidence. For UGP and UGPS, it was necessary to specify some BLs, therefore we defined the set of models M to use the following 11 models: Decision Tree (from now on, DT), Lasso Regression, Linear Regression (from now on, LR), Multi-Layer Perceptron (from now on, MLP), Random Forest (from now on, RF), Ridge Regression, Stochastic Gradient Descent (from now on, SGD) and Support Virtual Regression (from now on, SVR). MLP is present in 3 different variants, depending on the architectures used: MLP100 uses MLP with a single hidden layer composed of 400 units, MLP400 uses 400 units in its hidden layer and MLP200/100 has two hidden layers, each with 200 and 100 units respectively. Also, note that SGD is not a regression algorithm per se, but in this case it is used to performs a linear regression using Least Squares as a loss function. We used the models available in the scikit-learn Python package using the default settings and no tuning has been performed beforehand. The number of regression models available to the scientific community is of course much larger and not limited to the ones in scikit-learn: our choice was arbitrary and we select some of the most well-known and widely used models, trying to represent different classes of models, without being comprehensive.

The proposed method work with any underlying model

Note that not all the BLs are exactly "basic": among them there is Random Forest, which is a general purpose and widely used ensemble model which usually yields good results. We expect, on average, good performances from these BLs, therefore it is possible that the contribution brought by the evolutionary refinement will be marginal. Naturally, to be more consistent with the definition of BL, we could exclude complex models such as RFs or other ensemble learning methods from the initial population, but there is no practi-

Settings	bioav	concrete	energy	pkm	pkt
population size	100	200	200	200	200
G_s	200	200	200	20	20
G_l	2000	2000	2000	200	200
init strategy	RHaH	RHaH	RHaH	RHaH	RHaH
p_{xo}	0.5	0.7	0	0.7	0.7
p_{mut}	0.5	0.3	1	0.3	0.3
tournament size	4	4	4	4	4
mutation step	1	1	1	1	1
error measure	RMSE	RMSE	MAE	MAE	MAE

Table 4.2.1: Parameters for the evolutionary computations performed on the datasets under analysis. The population size was the same for all the datasets, as it was the initialization strategy used, ramped half-and-half. The number of generations varied depending on the dataset and the algorithms used. Tournament size and mutation step were fixed as well. Note that probability of crossover, p_{xo}, and mutation probability, p_{mut}, sum to 1.

cal reason for doing so, as UGP and UGPS are general meta-learning models that can exploit any kind of underlying method. There is no need to adopt truly basic learners in this context: leveraging arbitrarily complex models and build onto their result is precisely the purpose of this thesis.

We used parameters from original papers, with Standard GP functionals over the input variables

Settings for the GP algorithms are reported in Table 4.2.1. We tried to be as close as possible to the settings reported in the reference publications, but in some occasions we might have to adjust a parameter due to imprecision in the original papers, always after a confrontation with the original authors. Nevertheless, from the homogeneity of the settings, it is possible to see that the objective of these simulations was not to obtain the best possible results on the datasets, but to compare the behavior of the algorithms with existing literature. Therefore, we used the settings as-is without further tuning. The set of functionals used comprised the sum, subtraction, multiplication and protected division operators, while the set of terminals included the input variables only. The fitness function was Root Mean Squared Error between the target values and predicted outputs.

4.2.1 What was measured

For each performed test, we recorded the fitness value of the best individual of the population and its distribution. We also measured the semantic distance, running times, contribution of models and, when UGPS was used, the frequency of winning combinations in PESP. We will now describe in details the meaning of these measurements. The values are collected both on train (d_i^t) and test (d_i^v) datasets, if possible.

Measuring fitness is the foremost method to observe progress in finding solutions

Fitness value of the best individual is a widely used measure observed while running evolutionary algorithms. The best individual is determined at each generation as the individual which, in the current population, has the best fitness. The individual can (hopefully) change during the evolution, so this quantity summarize the general ability of the population to solve the problem and analyzing it through time helps understanding how much progress has been done and how easily a problem is getting solved. This is usually evident from the slope of the curve that, as it gets harder to improve the best solution, will reach a plateau (null derivative).

Of fitness values, we study values as well as sampling distributions

The distribution of the best fitness values is studied using only the last sampled value, meaning that, after each run, we use the fitness value of the best individual at the last generation as a sample and we observe how all these samples are distributed. This measurement allows us to understand the variance of the values or whether they assume a normal or non-normal distribution.

Studying the evolution of semantic distance we can see if evolution has impact on solutions

Semantic distance is an interesting metric that is calculated as the distance between the semantic vector of the best individual at each generation with the semantic vector of the best individual at initialization. More formally, if the sequence of best individuals at each generation is $i_0, i_1, i_2, \ldots$ and each of them will have a semantic vector $s_0, s_1, s_2, \ldots$, we collected the sequence of distances $D(s_0, s_0), D(s_0, s_1), D(s_0, s2), \ldots$ using $D = L^2$. This metric is interesting when studying UGP as it allows to see if the semantics evolves significantly during the evolution and therefore giving an understanding about how much the initial individuals generated from BLs are influenced by evolution.

We measure the impact of BL by counting the number of times they have been involved in the production of one individual

Models' contribution is a metric that gives us an insight into how much a model is contributing to the final solution. This metric is computed by associating to each individual a vector c that has length $|M| + 1$. Each component $c_j, j = 0, \ldots, |M|$ of this vector represents each of the models used as BLs

(plus one for the GP itself) and contains a value defined recursively: at generation 0 each individual generated randomly will have a c vector with $c_0 = 1$ and $\forall j! = 0, c_j = 0$, while individuals generated by the i-th BL will have a $c_i = 1$ and $\forall j! = i, c_j = 0$. From generation 1, each individual generated by crossover will have a vector defined as $c_j^x = c_j^p + c_j^q$, where c^p, c^q are the contribution vectors of the parents. For every other individual, be reproduced or mutated, the contribution vector will be the same as its parent individual. We then collect the contribution vector of the best individual during the evolution, in this way we can have an intuition of how much each model is contributing in the final solution.

Regarding UGPS, we also collected the frequency of selected models

In addition we counted, when UGPS was used, the frequency of winning model combinations. That is, during the PESP, the powerset of the models $2^{\mathcal{M}}$ is computed and each subset will be tested against a short run of UGP. Of the tested subsets, one is the best performing in terms of fitness. We count how many teams each subset is the best in each run. This highlights what are the "preferences" in term of models for each dataset, suggesting what kind of models are more appropriate to solve a certain problem.

Statistical tests have been run to test whether final fitness are sampled from different distributions

To ensure the statistical significance of the results regarding the final fitness values, that is, to ensure that the outcome of the algorithms is actually different and they are not samples from the same distribution, we performed different statistical tests. First, we performed a Lilliefors test to determine if the fitness values are normally distributed[55]. Then, a Kolmogorov-Smirnov two-distribution test (KS-test) was run to determine if two samples are drawn from the same distribution[55] and, finally, a Mann-Whitney-Wilcoxon test (also known as Mann-Whitney U-test or MWW test) was then run to determine if one distribution is stochastically greater than the other[60, 103]. KS-test and MWW test were run between all possible pairs of samples, that is, we compared GSGP outcomes with both UGP and UGPS, and we compared outcomes of UGP and UGPS as well.

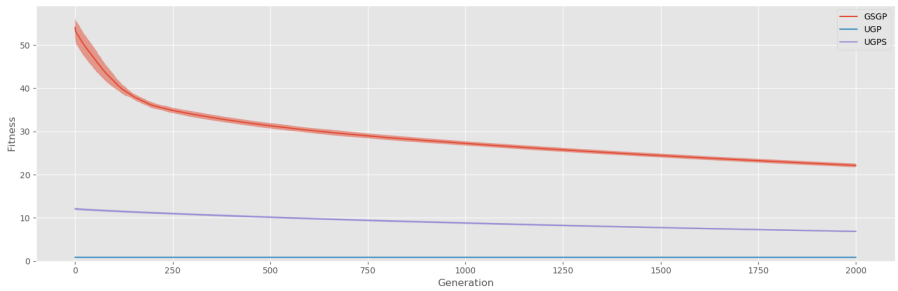

Figure 4.3.1: Fitness value of the best individual in the population at each generation for the bioav train dataset. These values are averaged for all the 60 runs performed, and reports both the median value that the median absolute dispersion for each curve. In this case, UGP appears to be the best performing algorithm.

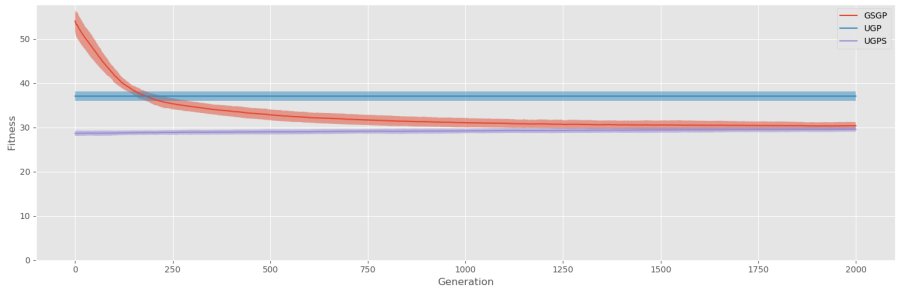

Figure 4.3.2: Fitness value for the best individual in the population for the bioav test dataset. The curves represent the median values of the 60 runs and to each curve is associated its median absolute dispersion. From this plot, we can see that UGP is likely overfitting, while UGPS is performing very well since the beginning of the evolution.

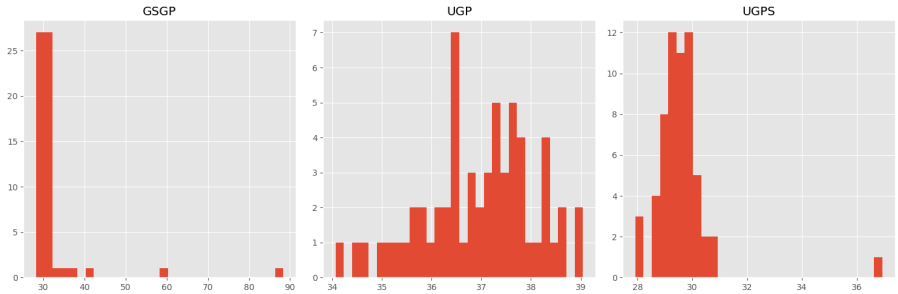

Figure 4.3.3: These histograms represent the absolute frequencies of the fitness values of the best individuals of the population over the 60 runs performed, for the test bioav dataset. As visible, GSGP suffers from a few outliers, but overall has a good final performance which is comparable to UGPS.

4.3 Results

We will start our discussion on the results by illustrating the bioav dataset, which is the one that suffered most from overfitting when using UGP and leads to the development of PESP in UGPS. Observe the evolution of the train fitness value against generations, in Figure 4.3.1: it is clear that UGP is having a significant advantage over GSGP and UGPS, given its low fitness since the beginning of the evolution. Compare also the starting point of GSGP with the starting points of UGP and UGPS: feeding the initial population with results of pre-trained models significantly improves the performances, as expected and, interestingly, the evolutionary process seems to improve the solutions, as it is visible from the UGPS curve. But since we are looking at training data, this behavior could also be due to overfitting: this explains why UGP is outperforming not only GSGP, but also UGPS: this should not normally happen, because UGPS is selecting the best models before evolving their solutions, and the only reason for having UGP performing better than UGPS, besides mere luck which should be ruled out by the 60 runs we performed, is that UGP includes an overfitting model, while UGPS has correctly discarded it. In fact, if we look at Figure 4.3.2 we can now see that UGP is performing worse than UGPS, differently from before, which is now the best performing of the three. It is also interesting to notice that UGPS slightly suffers from overfitting: the evolutionary process is excessively refining the initial solution and the fitness curve is slowly increasing in time. This phenomenon is natural if the model is not bounded and can grow in complexity, and letting it train indefinitely will lead to model learning the training data instead of learning to generalize.

Comparing these results with the ones in [99] we can see that our GSGP implementation is performing very similarly to the reported results, with a median value of about 20 in the training and 30 in the test sets. As in their work, our objective was not to obtain the best possible results on this dataset, therefore we used the same parameters settings when running the three algorithms. From this comparison we can see that UGPS outperforms the results previously obtained in literature with this kind of approach which, in turn, greatly improved the results over Standard GP.

As anticipated earlier, we run some statistical tests over the distribution of

Comparison	U statistic	MWW p-value	KS statistic	KS p-value
GSGP-vs- UGP	226	$1.472 \cdot 10^{-16}$	0.917	$1.867 \cdot 10^{-23}$
GSGP-vs- UGPS	2827	$7.135 \cdot 10^{-8}$	0.5	$2.813 \cdot 10^{-7}$
UGP-vs- UGPS	3572	$1.432 \cdot 10^{-20}$	0.983	$6.301 \cdot 10^{-27}$

Table 4.3.1: This table reports test statistics and p-values for the bioav dataset, comparing the final fitness values for each algorithm against the others. MWW U-test statistic is reported in the second column, with the associated p-value. From these data, we can assume that the U-test null hypotheses are not verified and therefore one distribution is stochastically greater than the other. In the fourth and fifth columns are reported the KS statistics and the associated p-values, suggesting that the samples come from two different distributions.

fitness values which revealed that test fitness values are likely not normally distributed (null hypothesis was rejected with $p < 0.001$). After running KS tests and MWW tests on last-generation fitness values, we reject the null hypothesis that data are drawn from the same distribution and that distributions are stochastically different. Results of the tests are reported in Table 4.3.1. Note that, having performed 60 runs, the maximum U value is 3600. This confirms our observations regarding the trends on the plots, but it is not to be excluded that the behavior of GSGP could approximate the one of UGPS as times increases.

To have a better picture of the evolution process, consider Figure 4.3.4, in which are shown the Euclidean distances between the semantics of the best individual of the first generation with the best individual of any subsequent generation. It is interesting to note the order of magnitude of change: without initializing the population using pre-trained models, semantic values have much more room to change, with semantics in GSGP changing almost ten times more than semantics in UGPS. Due to overfitting, this plot is basically a horizontal line for UGP, so we omit the plot.

It is also interesting to observe which models contributed more to the initial state of the system: since we are pre-training many models, it is natural to wonder if all of them are useful for the evolution, and what is their contribution to the evolutionary process. In Figure 4.3.5, top plot, we can see the contributions (normalized) of each model to the best individual for the entire

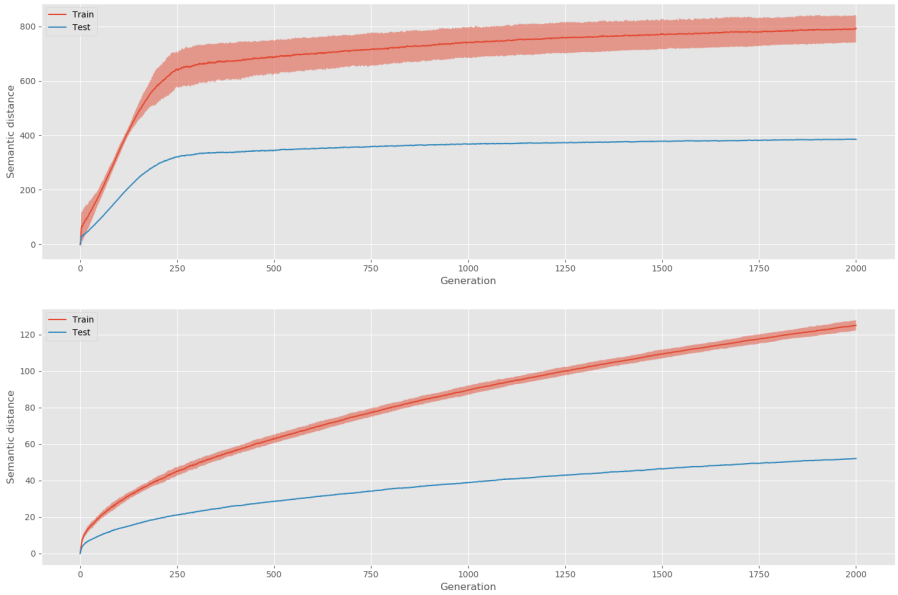

Figure 4.3.4: These plots represent the Euclidean distance, as evolution proceeds, between the semantic of the best individual at generation 0 with the semantic of the best individual in each other generation. The topmost plot shows the semantic distances for train and test datasets when GSGP is run, while the bottom plot shows the same for the UGPS algorithm. Semantic evolution for UGP is a flat line, due to overfitting, and was not reported.

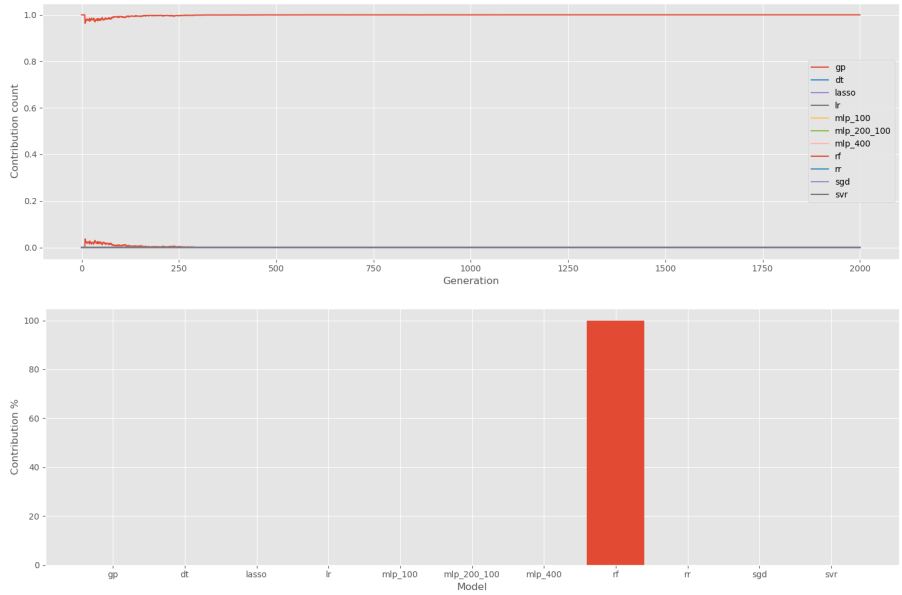

Figure 4.3.5: These plots shows the contribution of the models that took part into evolution. In the top one, the contribution is shown for each selected model during the evolution, while below it is shown the histogram of contributions at the last generation. These plots refers to models used by UGPS. In the case of UGP, the only contribution was brought by Decision Trees.

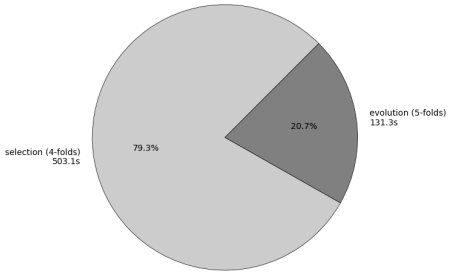

Figure 4.3.6: This pie chart shows the average running time for the UGPS algorithm. The light gray represents the time required to perform the selection of models, while the remaining part is relative to the long run. These times are averaged for the 60 runs, and it is visible that the strategy we adopted to select the models is clearly requiring a major computational effort. This is due not only to the number of models, but also to the fact that each model was run 20 times in total, due to nested folding cross validation.

evolution, while the histogram in Figure 4.3.5, on the bottom, shows the contribution frequency of each model at the last generation. It is noteworthy to observe how the main contributor was, almost exclusively, the Random Forest model: considering the overfitting phenomena that is being introduced by evolution, we deem possible that RFs are achieving a near-optimal prediction performance on this dataset.

Finally, let us conclude this section discussing the computational performances of the algorithms. In Figure 4.3.6 it is possible to see the amount of time required to run UGPS. As it is evident, the required amount to run the selection phase is very time consuming and it amounts for almost 80% of the required time. Wall-clock wise, UGPS took a total of about 10.5 minutes for each run, while UGP and GSGP required 2.5 and 1.7 minutes respectively. It must be noted, however, that the running time of UGPS is due to the strategy we adopted for the selection phase, which performs a nested K-fold cross-validation and is computationally very expensive. This is not necessarily needed in real-world applications, and adopting a more efficient strategy such as random train-validation split could is sufficient for most applications, drastically reducing the required computational effort. Consider that, to increase the statistical accuracy of these results, we nested a 4-fold cross-validation (for

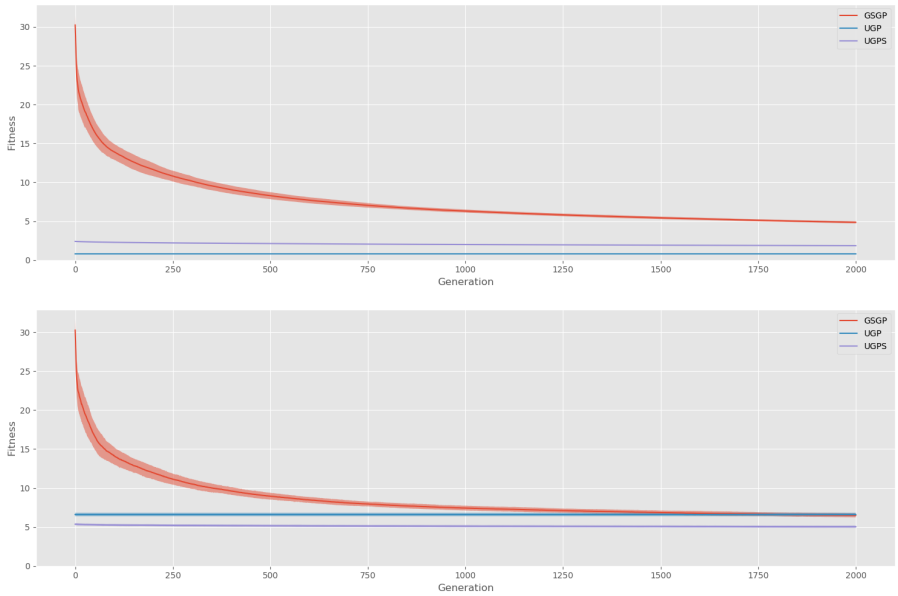

Figure 4.3.7: These plots show the distribution of fitness values for the three algorithms, both for train (above) and test (below) datasets. The represented median values are paired with median absolute dispersion of the data. As it happened with the bioav dataset, we can witness again to an overfitting phenomena.

each model) into a 5-fold cross-validation (for each dataset), meaning that we run UGP 20 times in total to perform a single run on a dataset. This is 4 times the effort of plain UGP, without PESP, in fact the computation time to perform the selection is roughly 4 times the evolution time (long run). If a faster strategy is adopted to search for the best models, the total running time of UGPS can be reduced significantly and brought on par with UGP.

The next dataset we used for testing is the concrete dataset. Let us start observing the train and testing fitness in Figure 4.3.7. We can see that GSGP improves its performances over time, both in training and testing datasets, while UGP and UGPS have almost a steady behavior, suggesting that initial models provided a good solution for this use case. Interestingly, UGPS is slightly decreasing over time in this case, differently from the bioav dataset, suggesting that evolution is still contributing to the optimization of the model. Note, however, that in the case of UGP the training performances are very good, while they are not so good during testing, possibly because one model is over-

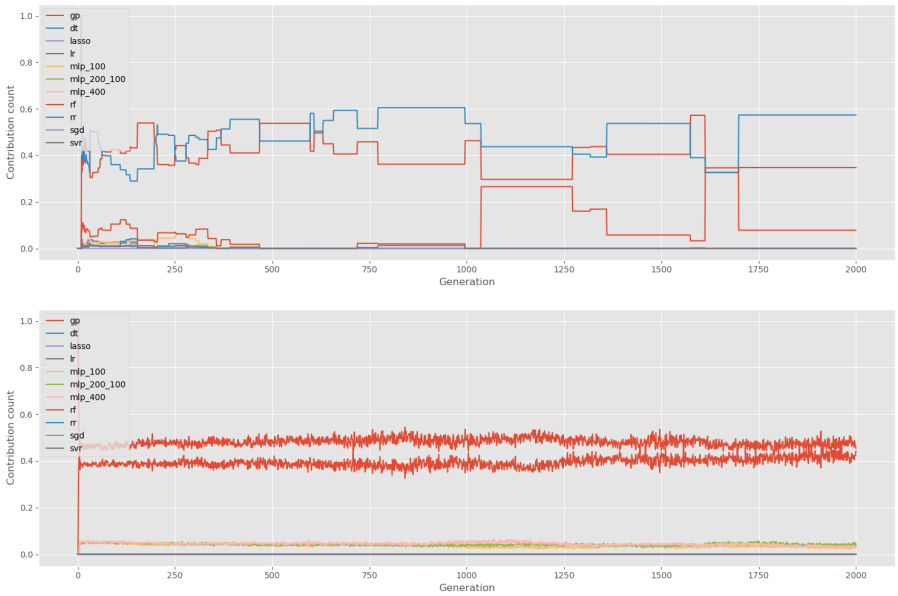

Figure 4.3.8: Above: the contribution, during the evolution of the UGP algorithm, of all the models. Below, the same plot, but for UGPS. Note how in UGP the contribution values are much less noisy than the ones in UGPS. We attribute this fact to Decision Trees being an overfitting solution which has been discarded by the selection procedure in UGPS.

fitting and was not excluded. This behavior is similar to what already happened earlier, supporting the fundamental role of PESP in UGPS.

Let us analyze now the models and their contributions. From this, we can understand if there is in fact a model which is prone to overfitting that was discarded by the selection procedure. Consider then Figure 4.3.8, in which contributions in UGP and UGPS can be compared. Note how, in UGP, the system is lead by 3 principal contributors: GP, DT and RF. If we compare this plot with the one of UGPS, we can see that GP and RF are still present, while DT has been excluded. Random Forest is an ensemble of Decision Trees: it is possible that, while a single DT is overfitting, its counterpart is not, therefore DT was excluded by PESP. This is reinforced by Figure 4.3.9, in which we can see that DT is never selected in a winning combination, even though it was the main contributor with UGP. Random Forests, on the other hand, are always present in any winning combination.

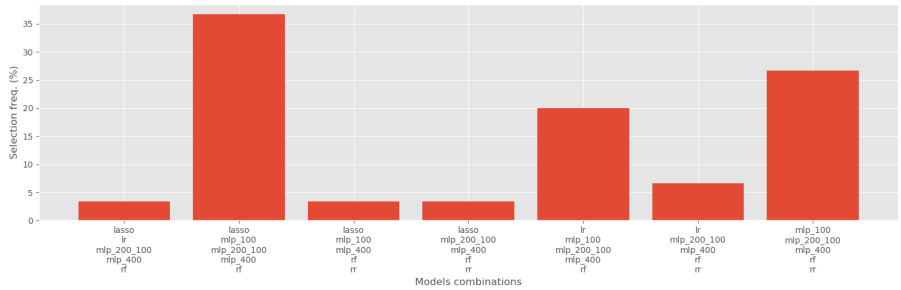

Figure 4.3.9: This histogram shows that some combination of models were selected more frequently by PESP, but note there is not a prevalent group. Interestingly, Decision Trees are never selected, but Random Forest are always selected, suggesting that the former might lead to overfitting solutions.

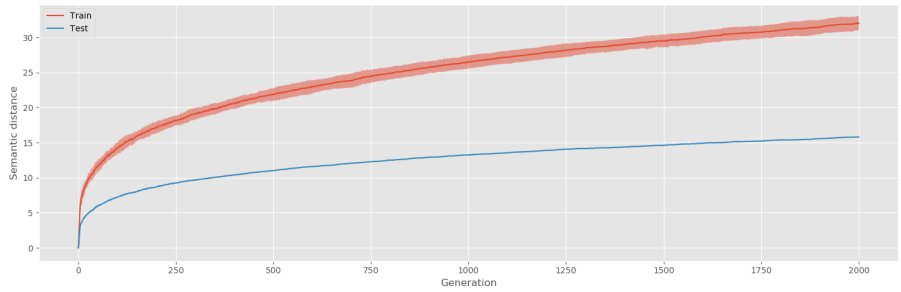

Figure 4.3.10: This figure shows evolution of semantic distance between the best individuals. Note that there is a progress in the evolution of the semantic, even if the initial solution turned out to be very good since the inception. We attribute this fact to the ability of GSGP to escape local optima and improve the solutions.

Comparison	U statistic	MWW p-value	KS statistic	KS p-value
GSGP-vs- UGP	1228	$2.703 \cdot 10^{-3}$	0.267	0.0224
GSGP-vs- UGPS	3600	$3.557 \cdot 10^{-21}$	1.0	$7.824 \cdot 10^{-28}$
UGP-vs- UGPS	3600	$3.557 \cdot 10^{-21}$	1.0	$7.824 \cdot 10^{-28}$

Table 4.3.2: This table reports MWW U-statistics and KS statistics for the fitness values of the best individuals obtained with each pair of algorithms. Note how in this case the evidence that GSGP and UGP are from different distributions is not very strong.

Interestingly, even if evolution is bringing a small contribution to the final outcome, in terms of fitness value, the distance of semantic vectors is changing significantly during UGPS evolution, as shown in Figure 4.3.10.

Differently from earlier, Lilliefors tests report that test data is likely normally distributed, having p-values for last-generation test values being 78%, 3% and 77% for GSGP, UGP and UGPS respectively. U and KS statistics are reported in Table 4.3.2, in which we can see that it is likely that GSGP and UGP samples are from the same distribution, but MWW test results that one variable is stochastically greater than the other. Therefore, also considering the trend of the values in Figure 4.3.7, we reject the null hypothesis that the samples are drawn from the same distribution. On the other hand, we exclude that samples of GSGP and UGPS are drawn from the same distribution, as visible both on plots and from the table.

When compared with results obtained in literature, in particular in [12], we can see a little discrepancy in the obtained results: while in the literature the final median training and testing error are 3.897 and 5.926 respectively, in our results we obtained a median of 4.855 and 6.499 for training and testing respectively. We are unable to explain the discrepancy in the results, as we used the same settings for the evolutionary computation. It is possible that this difference was due to the changes made to the original source code while porting it (although tests were performed to ensure no discrepancy), but we cannot state with certainty if this difference is due to such correction. We believe it is unlikely that such difference is due to the fact that previous results used a smaller number of runs (50 instead of 60) and that they used a randomized 70-30 split of the dataset instead of K-fold cross-validation. It is possible that discrepancies are caused by ill-reported error measure on the original paper.

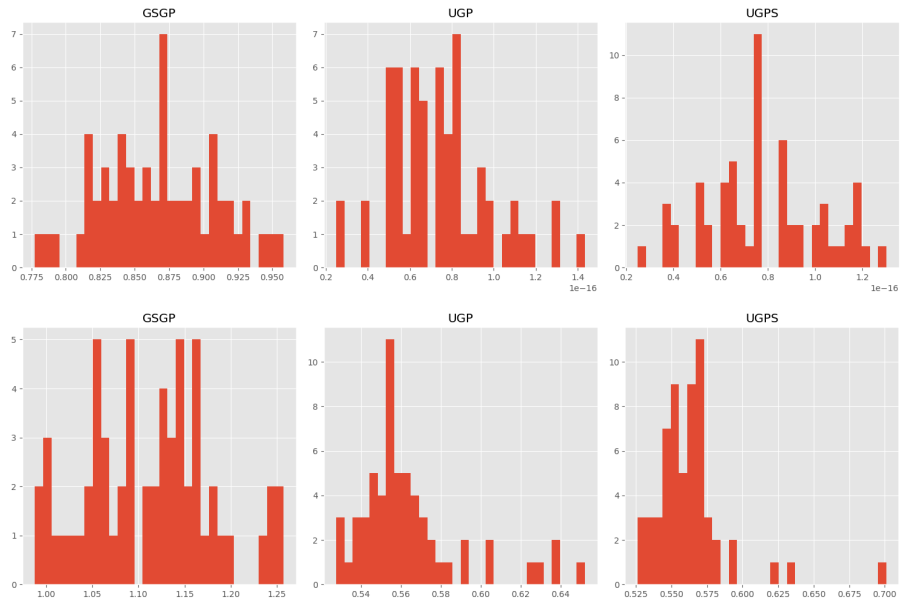

Figure 4.3.11: In these plots, we can see how UGP and UGPS perform against GSGP in terms of fitness values of the best individuals during the evolution. The new algorithms are performing very well on training data, scoring almost zero error, and are still significantly better than GSGP on the test dataset.

However, we can see that UGP and UGPS improved the results obtained in the literature, with the latter having a median fitness value of 1.856 and 5.036 for training and testing datasets respectively.

As earlier, the average running time for a single GSGP run was of about 329 seconds, 353 seconds for UGP and 1635 seconds for UGPS, of which about 1/5 is used by the long run, and 4/5 by the PESP. Again, the selection procedure emerged as necessary to avoid overfitting phenomena.

In the energy dataset, UGP and UGPS reach a remarkable performance, significantly improving the results obtained by GSGP. Consider Figure 4.3.11, in which we can see that the behavior of the two proposed algorithms is similar. In this case, none of the BLs appeared to overfit and the evolution process was non-influential, as shown in Figure 4.3.12, in which we can see that there is no improvement in fitness value. Differently from earlier, in this case the evolutionary process did not improve the semantic of the best individual in any way, as shown in Figure 4.3.13, in which semantic distance during evolu-

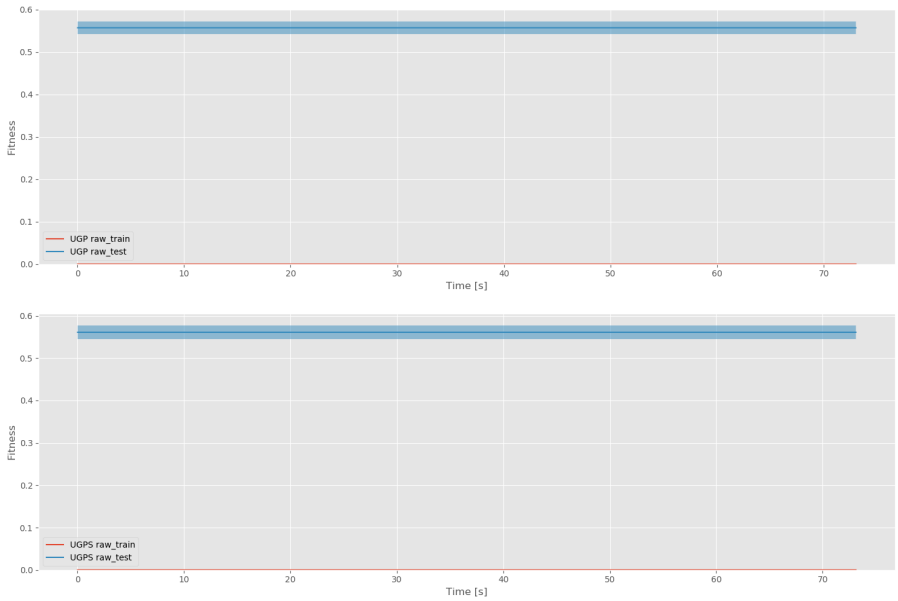

Figure 4.3.12: Above, the plot shows the evolution of the fitness values of the best individuals when using UGP. Below, the same plot for UGPS. Note how the train and test values are stable, without increasing or decreasing over time. This likely means that the Base Learners found an optimal solution and that GSGP is not able to improve upon it, not even by overfitting.

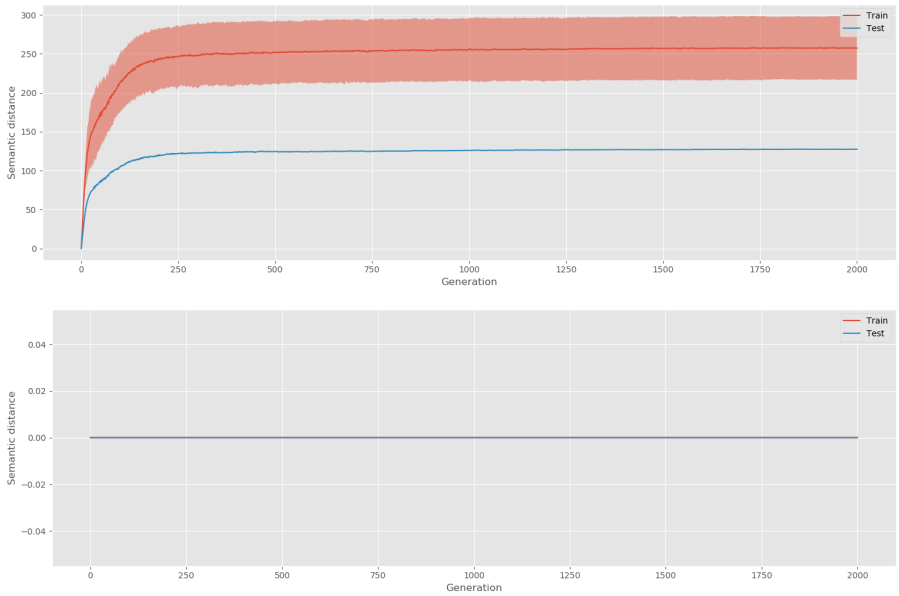

Figure 4.3.13: In this figure, plots of distance evolution of GSGP (above) and UGPS (below) are compared. Note how, while GSGP is actually working on the transformation of semantic values over the evolution, UGPS has never changed the semantic of the best individual. This is likely due to the elitism mechanism, which kept a solution from a basic learner into its place, with no chances to alter it. This is due to the fact that such solution is an excellent result, possibly not far from being a global optima.

tion of GSGP is compared to the one of UGP (UGPS is behaving like UGP in this case, so we omit the plot).

This behavior could mean that, on this dataset, the BL has reached a solution very near the global optimum, leaving no room for improvement for the evolutionary process. It is interesting to see that, even in this case, Decision Trees played an important role in the outcome, being the only contributor to the entire evolution. It is interesting to notice the combination of models that won more often during PESP: differently from the previous cases, when different combinations of models were chosen for the long evolution process, in this case a single combination won 59 times out of 60. This group was composed of 5 models: DT, RF, SVR and MLP in the 200/100 and 400 variants. In the single run in which this combination did not win, the winning combination was the same, but LR was selected instead of MLP400.

Comparison	U statistic	MWW p-value	KS statistic	KS p-value
GSGP-vs- UGP	3600	$3.557 \cdot 10^{-21}$	1.0	$7.824 \cdot 10^{-28}$
GSGP-vs- UGPS	3600	$3.557 \cdot 10^{-21}$	1.0	$7.824 \cdot 10^{-28}$
UGP-vs- UGPS	1780	0.918	0.1	0.911

Table 4.3.3: This table reports U-statistics and KS statistics, as well as associated p-values, for the fitness values of the best individuals obtained with each pair of algorithms. p-values for the comparison of UGP and UGPS are very high, meaning that it is extremely likely that the null hypothesis is verified, that is, that fitness values for UGP and UGPS were drawn from the same distribution. This leads us into rejecting the alternate hypothesis.

We do not report contribution plot, in this case, as they are flat lines from the first to the last generations, as GP was unable to improve the results, not even by overfitting.

Regarding normal distribution, for this dataset we obtain from Lilliefors that only GSGP is likely to be normally distributed, with a p-value of 75%, while p-value is less than 0.001 for the values of other algorithms. From Table 4.3.3 we can see that it is extremely likely that GSGP values have been sampled from a distribution which differs from the one in which UGP and UGPS samples were sampled. Specifically, it is instead very likely that values for these two algorithms have been sampled from the same distribution. This is not surprising, because in this case no overfitting phenomenon was recognized and the outcome of both algorithms was influenced by a single BL and statistical tests confirm our thesis.

When comparing to literature results in [15], we compare our results with the first target variable, Y1 (heating load). Again, we are unable to precisely reproduce the results from the paper, obtaining a slightly larger error. Nevertheless, the results from GSGP are compatible, and again UGP and UGPS reduced significantly the errors, almost halving it. It is interesting to notice that our algorithms perform very similarly to the new algorithms proposed in the paper, namely HYBRID and HYBRID-LIN, which adopt Linear Scaling[40] and local optimization techniques to improve upon basic GSGP. In our case, Decision Trees played a key role in providing the best solution: due to their nature, they are insensitive to translation and scale, therefore modifications

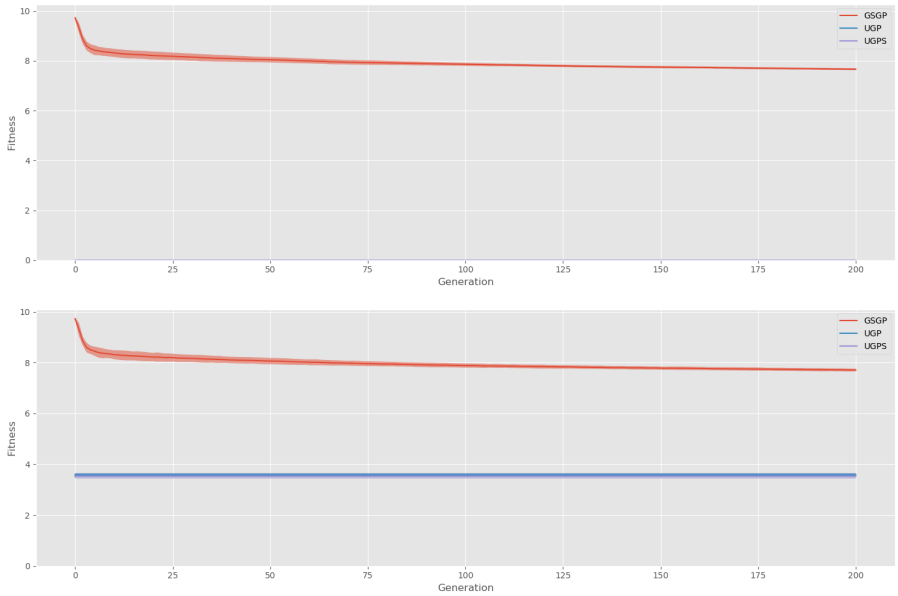

Figure 4.3.14: These plots show fitness values of the best individuals during the evolution. As visible in the plot above, the fitness value in UGP and UGPS is almost zero, meaning that the algorithm can model the training data almost with no errors. Below, we can see that the error on test data is not zero, as expected, but it is not a case of overfitness, as the algorithms are performing much better than GSGP, even if evolution did not bring any significant contribution in the final results.

such as Linear Scaling would have little or no impact on their performances.

Regarding the processing times, average wall-clock time of a single run of GSGP took 381 seconds, 403 seconds for UGP and 389 seconds for the long run of UGPS. Again, the selection timing was sensibly more expensive, being almost 1493 seconds, almost 4 times more than the long run, as expected.

We now describe the pkm and pkt datasets, which behaved similarly. We begin describing the behavior of pkm and terminate with pkt, but without including the figures for this last one, due to their similarities to the ones of pkm. As earlier, the pkm dataset shows that BL reached a near-zero error on the training dataset and halved the error in the testing dataset when comparing to GSGP. Both UGP and UGPS performed well, with the latter performing slightly better on average, as shown in Figure 4.3.14. There no sign of overfitting in any of the three algorithms. As already happened, there is no significant improvement in the fitness value brought by evolution, but if we observe

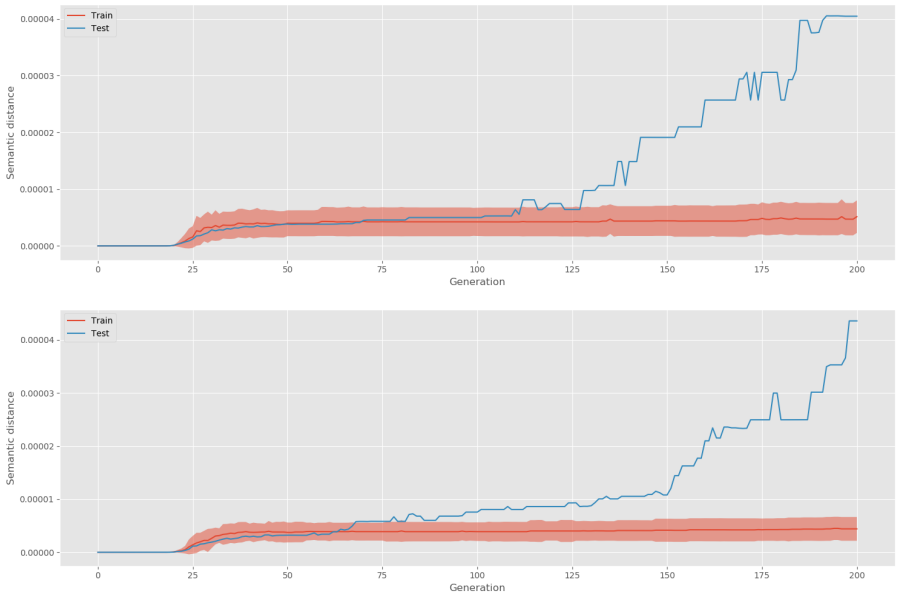

Figure 4.3.15: In this plot it is visible the evolution of the semantic distance between the best individuals at each generation since the first generation. Note how, both in UGP and UGPS, above and below, respectively, evolution could slowly change the semantics. This appears to be a case in which GSGP escaped a local optima and could find another area to exploit in the fitness landscape. Interesting to note how the semantic distance, albeit on a very small scale, has its major shift in the test dataset only.

the semantic distance of UGP and UGPS, in Figure 4.3.15, we can see that evolution actually changed the solution, albeit the variation is very small if compared to the one of GSGP.

Again, Decision Trees are the preferred model, as it can be seen in Figure 4.3.16 in which we can see that GP and DT are the most contributing models, along with the presence of RF.

The behavior with these datasets is similar even in the results of the statistical tests performed. Lilliefors report that, both in pkm and pkt, final fitness values are likely normally distributed, with p-values never below 2%. In addition, as happened with the energy dataset, we can see that resulting values from UGP and UGPS are very likely drawn from the same distribution, while

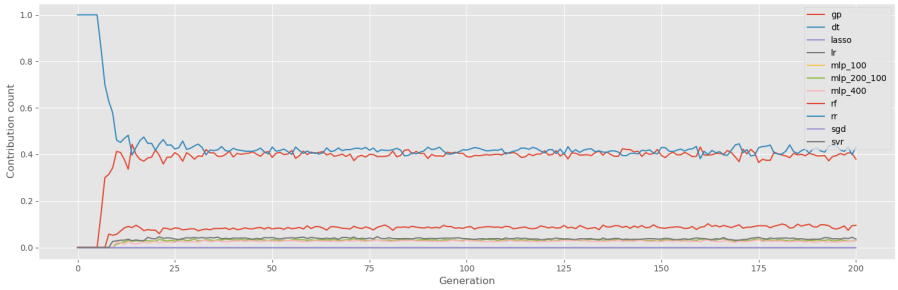

Figure 4.3.16: This plot shows the contribution brought by selected models during UGPS evolution. Note how Genetic Programming and Decision Trees are the main contributions, strictly tied to each other. Random Forest scores also a relevant contribution during the evolution.

Comparison	U statistic	MWW p-value	KS statistic	KS p-value
GSGP-vs- UGP	3600	$3.557 \cdot 10^{-21}$	1.0	$7.824 \cdot 10^{-28}$
GSGP-vs- UGPS	3600	$3.557 \cdot 10^{-21}$	1.0	$7.824 \cdot 10^{-28}$
UGP-vs- UGPS	2050	0.190	0.266	0.022

Table 4.3.4: Table reporting the results of MWW and KS tests. Fitness values of the best individuals at the last generation were considered, for each pair of algorithms. p-values for the comparison of UGP and UGPS are reasonably high, that is, it is likely that the samples drawn by UGP and UGPS are from the same distribution. Therefore, we reject the alternate hypothesis.

Comparison	U statistic	MWW p-value	KS statistic	KS p-value
GSGP-vs- UGP	3600	$3.557 \cdot 10^{-21}$	1.0	$7.824 \cdot 10^{-28}$
GSGP-vs- UGPS	3600	$3.557 \cdot 10^{-21}$	1.0	$7.824 \cdot 10^{-28}$
UGP-vs- UGPS	1679	0.527	0.183	0.239

Table 4.3.5: Report of the values produced by MWW U-test and KS test, including the value of the statistics as well as the associated p-values. Here, the fitness values of the best individuals at the last generations were tested, and we compared the values obtained by each pair of algorithms. Since for the last combination we observe high p-values, we accept the null hypotheses, that the samples obtained by UGP and UGPS are from the same distribution.

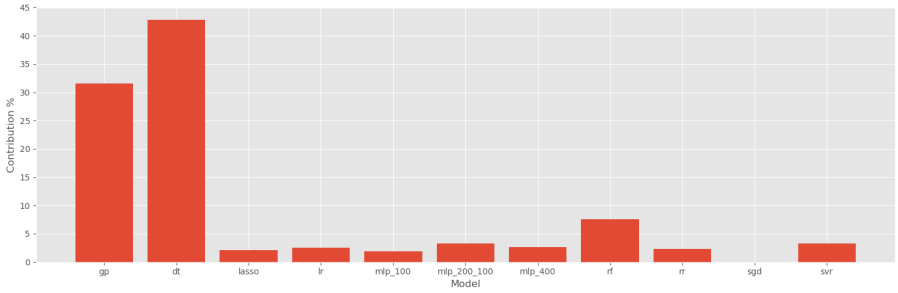

Figure 4.3.17: This histogram shows the average relative contribution of each model at the last generation of UGP. In this case, we can see that Decision Trees and Genetic Programming are contributing almost exclusively to the evolution, altgough Random Forest has also a minor role in this process, influencing about 7% of the final best solution.

the same does not hold for samples extracted from GSGP, and this was visible in the plots shown above. This is likely due to the fact that both UGP and UGPS are heavily dependant on DT and GP basic learners, in both Parkinson datasets, and with no apparent overfitting, the behavior of the two algorithms is very similar.

Differently from earlier, the computational times were noticeably in favor of GSGP, which required, on average, 129 seconds per run, while UGP required 342 seconds, UGPS 306 seconds in the average long run and 1105 seconds for an average selection process.

pkt behaved very similarly, and the same considerations hold. Exactly as pkm, the winning combination of models was a single one, composed by DT, RF, SVR and MLP, in the 200/100 and 400 variants. The time required for a single run was reduced as well, being 126 seconds on average for GSGP, 331 seconds for UGP and 296 seconds for UGPS in addition to 1057 seconds for the selection. Contribution by models is also very similar among the two, with GP contributing slightly more in pkt than pkm, but still accounting to a little less than 40% of the final contribution, while DT contributes in both cases a little more than 40%, and the remaining contribution has been brought by the other models involved, especially RF.

Not surprisingly, during UGP almost all the models contributed, even if marginally, to the final result, as reported in Figure 4.3.17, but without affecting the contribution by DT, while the model that was mostly affected was GP.

Likely, this is because the presence of more models, all of which (except of SGD) could be meaningful to the evolution process, removing room for the evolution of pure-GP solutions.

When comparing our results to literature[13], our implementation of GSGP is yielding, differently from earlier, non-trivial discrepancies in the train and testing errors. We are unable to reproduce precisely their results. In this case, we are noticing significantly larger errors when compared to results in literature: while their median error for GSGP is about 5.5 for the pkm test dataset and about 7 for the pkt test dataset, our errors are around 7 and 10, respectively. This is, of course, taking into account the different error measure, as reported in the paper, which is Mean Absolute Error instead of Root Mean Squared Error. Nevertheless, even if the errors we report are significantly larger, there is room for improvement and UGP and UGPS perform very well, reducing the error of about 54% when compared to GSGP, on the pkm dataset, and about 58% on the pkt dataset, shifting the test fitnesses from the values stated earlier to 3.54 and 4.29 on each respective testing dataset.

5

Conclusion

Our objective was to approximate a function whose semantic is known in some points

Given a set of regression models $\mathcal{M}$ and a dataset $\mathcal{D} = (X, y)$, where X is the set of input instances and y is the target output for each instance computed by an unknown function $\hat{f}$, the semantic of those models is the vector of their output when evaluated on that dataset, that is, $\hat{y} = f_i(X)$ for each model $f_i \in \mathcal{M}$. The semantic vector is therefore a point in the space of semantics $\hat{y} \in \mathbb{R}^{|\mathcal{D}|}$, and the objective of each regression algorithm is to minimize the distance $f_i(X) - y$.

GSGP evolves solutions in a way that have known effects on their semantics

Geometric Semantic Genetic Programming is an evolutionary algorithm for regression that has proven to effectively produce good approximations of the objective by evolving, though the principles of natural evolution, a set of randomly generated models (called individuals) and their semantics. Those are combined in a way that exploits their geometric nature to reach a very good approximation of the objective semantic y. In particular, exploiting the geometric space it is possible to modify and combine the individuals in a way that ensure that the semantics of their combinations are no worse than any of the original semantics. In this setting, the fitness landscape of the problem is uni-

modal and under some sense easier to explore than the space of solutions.

Leveraging the nature of semantics we initialized the GSGP population with pre-trained models

In this thesis, we exploited the fact that semantics for a given dataset are homogeneous independently from the model they are generated from; in the case of GSGP, every individual has the same nature, but their semantics are points in the same space of the semantics generated by models in $\mathcal{M}$ on the dataset $\mathcal{D}$. Therefore, we hypothesized that by producing semantics using Machine Learning models pre-trained over the same dataset and including them in the evolutionary process could significantly improving the performances, as good approximating solutions would be part of the initial population and, by using evolution, those could possibly be improved. We refer to this modification of GSGP which uses externally pre-trained models as Universal Genetic Programming, to denote its ability to work with any kind of model.

Seeded solutions must be properly selected or overfitting is a likely outcome

Such hypothesis works when training, but it is possible that the resulting solutions are not able to generalize on unseen data. This is valid in general for any Machine Learning model, but it is a very concrete possibility the case of UGP, because if any of the pre-trained model is overfitting, its training accuracy will be very high and it will impact negatively the evolution of every other individual. Experimental results confirmed this and showed that UGP is occasionally unable to generalize at all and, in general, is very sensible to overfitting models. For this reason, we introduced a pre-evolutionary selection procedure, referred as PESP, which attempts to exclude from the evolution the models which are unable to generalize. To do so, we have to train the algorithms on some data and evaluate them on some unseen data. To determine which models have to be included or not, we proposed to evaluate their performances as-if they were included in a run of UGP: one or more models can be included in a short "preview" run of UGP and, once a short evolution is concluded, the output performances can be evaluated on unseen validation data. Initially, we evaluated the performances of every possible combination of models, yielding interesting insights on the possibly synergistic behavior of multiple models that co-evolved. This approach, although exhaustive, is also computationally very expensive and scales badly for an high number of models, being the number of possible combinations $2^{|\mathcal{M}|}$. Therefore, we tested and showed in this thesis the case in which every model is evaluated by itself, again by performing a short run of evolution and including it in the initial pop-

ulation, and we included the 50%-best models in the initial population for a "long" evolutionary run. This approached turned out to be functioning rather well and we refer with UGPS to the method of running UGP preceded by PESP.

We tested our algorithms over 5 known dataset and compared the results with state of the art

We then applied the proposed algorithms, UGP and UGPS, along with the base algorithm GSGP, to five different datasets that were previously used to test the performances of GSGP. These datasets referred to problems of various nature: estimating the oral bio-availability of a drug, predicting performances of high-performance concrete given its composition, determine the consumption of a building given some of its properties and estimating the severity of the symptoms of a patient with Parkinson's disease.

UGPS properly selected and evolved the best models in the initial population and results were improved

As it is possible to observe in 4, results are positive and encouraging. The algorithms behaved as expected, in the sense that GP improved, where possible, upon the results of BLs. With the dataset we analyzed, few models proved to be working very well, in particular Decision Trees and Random Forest. This limited the role of GP in finding the best solution, but did not hinder the capacity of the system as a whole. The outcome is positive, improving significantly upon the results obtained by GSGP. Indeed, following these results there are a few considerations to be made, some of which were anticipated earlier and which are related to the computational effort required to obtain such results.

We conclude that this path is viable and fertile terrain for further improvements

Results are encouraging, showing that the concept is valid and the path viable: we have shown that UGP and UGPS are a potential improvement over GSGP with respect to the quality of the results in terms of fitness value, i.e. objective function error. Therefore, these methods can, in theory, be used to solve problems where higher precision is required. In particular, introducing a pre-selection strategy that filtered out overfitting models proven to be an effective solution that stood up to the challenge posed by these datasets: without it, UGP failed when compared to GSGP, in few cases, due to overfitting in the initial population. Therefore, we exclude that UGP could be used safely without a preventive selection strategy and we deem UGPS as the viable and effective alternative to GSGP.

5.1 Discussion

The adopted strategies were purposely time-consuming and leaner options must be considered

Considering that training the adopted BLs requires a fraction of the time required to improve the solutions via evolutionary computation, one might object that this effort is excessive to obtain the results, especially after seeing that few BLs are capable of obtaining excellent outputs. This is true. Nevertheless, there are some options that must be considered before discarding the proposed algorithms. First, we adopted a PESP which is computationally very expensive for the sake of precise and statistically significant results. There is no need to perform this kind of computation in every situation, and in fact there are alternatives which are computationally lighter. If the dataset is large enough, it is indeed possible to avoid K-fold cross-validation and still obtain a statistically meaningful result. We adopted K-fold cross-validation on two different levels, building a "nested" KJ-fold validation which required a significant amount of computational effort, which was applicable in this case thanks to the generally good performances of the parallel implementation. Therefore, instead of using a nested folded cross-validation, a three-way random split of the dataset could be used as well, partitioning it in train, validation and test datasets. This strategy is solid and often used by Machine Learning practitioners and the statistical significance of the results can be obtained by running the algorithms multiple times with different random splits.

Early stopping is needed to reduce the computational time and the risk of overfitting

This is not the only improvement that can be applied to reduce unnecessary computational effort. In fact, it is possible to see in the plots shown in Chapter 4 that the evolution continued for a long time, even if the solution was stable and the semantic edit performed by the GP was negligible. We adopted this strategy to compare exactly with the literature results of GSGP, and it was required to let GSGP evolve up to a prefixed point in time, but, in practice, this is not required at all. There are in fact widely diffused strategies to terminate the evolution process without having to wait the maximum number of generations (for an overview, consider [83]). These strategies rely on the behavior of the fitness value in time, for example, if the fitness does not improve significantly for a certain number of generations, the evolution could be stopped. In addition, it is possible to devise new strategies which do not only rely on the behavior of the fitness value, but also on the progress of the semantic distance.

Therefore, with no changes in the implementation of the algorithm, but just a different strategy when running tests, it is possible to dramatically improve the required time of computation, transforming UGPS into a viable alternative to GSGP and an effective ensemble method which can be used to solve real-world problems.

Other techniques to reduce overfitting and improve evolvability should be tested to make the algorithm more robust

Other improvement points are relative to Geometric Semantic Operators and additional overfitting control. As we saw, there are different techniques that can be used to reduce the risk of overfitting. Early stopping is one of them and we proposed it above. Other techniques involve the use of different subsets of $\mathcal{D}$ during the evolution, so that it is harder for an individual to overfit a specific training dataset due to its variability, and to use neighborhood-informed evaluation of the efficacy of the evolution to determine the stop of the evolution [33]. Regarding the semantic operators, a number of improvements have been proposed, for example [100] or, as we mentioned, re-evaluating the presence of the crossover operator [16], and it is indeed worthy to explore the effects of such improvements in the context of UGPS, on even more datasets, possibly improving the performances of the evolutionary search.

The methods are effective in selecting a good subset of models

The ensemble and meta-learning nature of these algorithms shall not be overlooked: the ability to effectively chose among different models and improve upon them is very useful nowadays, where a high number of models and methods exist to solve problems, and an automated method that can reliably select which ones can be used is indeed valuable. In addition, thanks to the "intermediate representation" provided by the semantic, any model can be adopted as a BL, and this feature is not common among neither Ensemble or Meta-Learning Methods. We have shown that the use of semantics in the context of meta-learning is a viable and effective concept.

More models, even automatically generated, would push the scaling capabilities of these algorithms

A larger number of BLs would be desirable in practical applications, but this would, of course, require a much larger effort when it comes to exclude and penalize overfitting models: training many models is indeed a significant effort, but testing combinations of models or even evaluating the performances of each model might be prohibitive if a short run of GSGP is run for each of them, as we did. In addition, there are meta-models (such as Neural Networks) that depends on some parameters and being able to automatically generate them

as part of the algorithm would be an even more effective contribution, but its effectiveness would be even more tied to the selection procedure. For this reason, the short run of GSGP could be removed entirely and different validation strategies could be applied. In case of a large number of individuals coming from BLs, it is more appropriate to develop a selection of the best, non-overfitting models during the evolution. This would allow to exploit the presence of all the models, filtering the less worthy when appropriate.

The ratio of random-vs-seeded models should be studied to unveil possible new behaviors

Also, the relation between the fraction of population that is seeded and the fraction that is randomly initialized should be investigated. It is possible that a large portion of pre-trained models will impact negatively the randomly generated individuals: the selection pressure will be in favour of stronger individuals, basically removing random individuals from the population in a few generations. From this consideration, two possible research paths could stem: one is the investigation of an evolution with no random individuals at all. In this case, GSGP would exclusively act as an "aggregator", which combines and weights existing solutions trying to improve them. On the other hand, it is interesting to give random individuals another chance and let them to evolve for some time before introducing the pre-trained individuals. In other words, the effects of BLs would be delayed in time.

Multi-objective and final model synthesis are features that could improve the practical utility of UGPS

Finally, we think that the implementation could benefit from some changes that might indeed be not only interesting from a research point of view, but also on a practical side. One change is the implementation of multi-objective search. In addition, we completely overlooked the step of yielding the actual best individual that evolution has produced. To do so, it is of course necessary to include the semantics of the BLs in the solution, therefore a final individual would be composed by a subset of $\mathcal{M}$ and a tree that transforms its semantics to the final solution. This feature is particularly important for real-life applications where the interest is to apply the model to predict certain behaviors and not only to evaluate the potential performances of the algorithm, as we did in this work.

References

[1] Ricardo Aler, Daniel Borrajo, and Pedro Isasi. Using genetic programming to learn and improve control knowledge. *Artificial Intelligence*, 141(1-2):29–56, 2002.

[2] Francesco Archetti, Stefano Lanzeni, Enza Messina, and Leonardo Vanneschi. Genetic programming for human oral bioavailability of drugs. In *Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 255–262. ACM, 2006.

[3] Francesco Archetti, Stefano Lanzeni, Enza Messina, and Leonardo Vanneschi. Genetic programming for computational pharmacokinetics in drug discovery and development. *Genetic Programming and Evolvable Machines*, 8(4):413–432, 2007.

[4] Wolfgang Banzhaf and William B. Langdon. Some considerations on the reason for bloat. *Genetic Programming and Evolvable Machines*, 3 (1):81–91, 2002.

[5] Wolfgang Banzhaf, JR Koza, C Ryan, L Spector, and C Jacob. Genetic programming. *IEEE Intelligent Systems and their Applications*, 15(3): 74–84, 2000.

[6] Tobias Blickle and Lothar Thiele. A mathematical analysis of tournament selection. In *ICGA*, pages 9–16. Citeseer, 1995.

[7] Markus F Brameier and Wolfgang Banzhaf. *Linear genetic programming*. Springer Science & Business Media, 2007.

[8] Leo Breiman. Bagging predictors. *Machine learning*, 24(2):123–140, 1996.

[9] Leo Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[10] Edmund K Burke, Michel Gendreau, Matthew Hyde, Graham Kendall, Gabriela Ochoa, Ender Özcan, and Rong Qu. Hyper-heuristics: A survey of the state of the art. *Journal of the Operational Research Society*, 64(12):1695–1724, 2013.

[11] Weihua Cai, Arturo Pacheco-Vega, Mihir Sen, and Kwang-Tzu Yang. Heat transfer correlations by symbolic regression. *International Journal of Heat and Mass Transfer*, 49(23-24):4352–4359, 2006.

[12] Mauro Castelli, Leonardo Vanneschi, and Sara Silva. Prediction of high performance concrete strength using genetic programming with geometric semantic genetic operators. *Expert Systems with Applications*, 40(17):6856–6862, 2013.

[13] Mauro Castelli, Leonardo Vanneschi, and Sara Silva. Prediction of the unified parkinson's disease rating scale assessment using a genetic programming system with geometric semantic genetic operators. *Expert Systems with Applications*, 41(10):4608–4616, 2014.

[14] Mauro Castelli, Sara Silva, and Leonardo Vanneschi. A c++ framework for geometric semantic genetic programming. *Genetic Programming and Evolvable Machines*, 16(1):73–81, 2015.

[15] Mauro Castelli, Leonardo Trujillo, Leonardo Vanneschi, and Aleš Popovič. Prediction of energy performance of residential buildings: A genetic programming approach. *Energy and Buildings*, 102:67–74, 2015.

[16] Mauro Castelli, Luca Manzoni, Ivo Gonçalves, Leonardo Vanneschi, Leonardo Trujillo, and Sara Silva. An analysis of geometric semantic crossover: A computational geometry approach. In *IJCCI (ECTA)*, pages 201–208, 2016.

[17] Mauro Castelli, Ivo Gonçalves, Leonardo Trujillo, and Aleš Popovič. An evolutionary system for ozone concentration forecasting. *Information Systems Frontiers*, 19(5):1123–1132, 2017.

[18] Mauro Castelli, Luca Manzoni, Sara Silva, Leonardo Vanneschi, and Aleš Popovič. The influence of population size in geometric semantic gp. *Swarm and Evolutionary Computation*, 32:110–120, 2017.

[19] Mauro Castelli, Raul Sormani, Leonardo Trujillo, and Aleš Popovič. Predicting per capita violent crimes in urban areas: an artificial intelligence approach. *Journal of Ambient Intelligence and Humanized Computing*, 8(1):29–36, 2017.

[20] Shu-Heng Chen, John Duffy, and Chia-Hsuan Yeh. Equilibrium selection via adaptation: Using genetic programming to model learning in a coordination game. In *Advances in Dynamic Games*, pages 571–598. Springer, 2005.

[21] Eric Collom. Applying genetic programming to bytecode and assembly. *Scholarly Horizons: University of Minnesota, Morris Undergraduate Journal*, 1(2):4, 2014.

[22] Joao Correia, Penousal Machado, Juan Romero, and Adrian Carballal. Evolving figurative images using expression-based evolutionary art. In *Iccc*, pages 24–31, 2013.

[23] Charles Darwin. *On the origin of species, 1859*. Routledge, 2004.

[24] Edwin D De Jong, Richard A Watson, and Jordan B Pollack. Reducing bloat and promoting diversity using multi-objective methods. In *Proceedings of the 3rd Annual Conference on Genetic and Evolutionary Computation*, pages 11–18. Morgan Kaufmann Publishers Inc., 2001.

[25] Pedro Domingos. *The master algorithm: How the quest for the ultimate learning machine will remake our world*. Basic Books, 2015.

[26] Chris Drummond. Replicability is not reproducibility: nor is it good science. 2009.

[27] Agoston E Eiben, James E Smith, et al. *Introduction to evolutionary computing*, volume 53. Springer, 2003.

[28] David B Fogel, Lawrence J Fogel, and J Wirt Atmar. Meta-evolutionary programming. In *Signals, systems and computers, 1991. 1991 Conference record of the twenty-fifth asilomar conference on*, pages 540–545. IEEE, 1991.

[29] Lawrence J Fogel, Alvin J Owens, and Michael J Walsh. Artificial intelligence through simulated evolution. 1966.

[30] Edgar Galván-López, James McDermott, Michael O'Neill, and Anthony Brabazon. Defining locality as a problem difficulty measure in genetic programming. *Genetic Programming and Evolvable Machines*, 12(4):365–401, 2011.

[31] David E Goldberg and Kalyanmoy Deb. A comparative analysis of selection schemes used in genetic algorithms. In *Foundations of genetic algorithms*, volume 1, pages 69–93. Elsevier, 1991.

[32] Ivo Gonçalves, Sara Silva, and Carlos M Fonseca. On the generalization ability of geometric semantic genetic programming. In *European Conference on Genetic Programming*, pages 41–52. Springer, 2015.

[33] Ivo Gonçalves, Sara Silva, Carlos M Fonseca, and Mauro Castelli. Unsure when to stop?: ask your semantic neighbors. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 929–936. ACM, 2017.

[34] Ivo Carlos Pereira Gonçalves. *An exploration of generalization and overfitting in genetic programming: standard and geometric semantic approaches*. PhD thesis, Universidade de Coimbra, 2017.

[35] Simon Harding and Wolfgang Banzhaf. Fast genetic programming on gpus. In *European Conference on Genetic Programming*, pages 90–101. Springer, 2007.

[36] Ami Hauptman and Moshe Sipper. Evolution of an efficient search algorithm for the mate-in-n problem in chess. In *European Conference on Genetic Programming*, pages 78–89. Springer, 2007.

[37] John H Holland. Genetic algorithms and the optimal allocation of trials. *SIAM Journal on Computing*, 2(2):88–105, 1973.

[38] Paul Holmes. The odin genetic programming system technical report rr-95-3 computer studies department napier university. 1995.

[39] Ting Hu and Wolfgang Banzhaf. The role of population size in rate of evolution in genetic programming. In *European Conference on Genetic Programming*, pages 85–96. Springer, 2009.

[40] Maarten Keijzer. Improving symbolic regression with interval arithmetic and linear scaling. In *European Conference on Genetic Programming*, pages 70–82. Springer, 2003.

[41] Kenneth E Kinnear. Evolving a sort: Lessons in genetic programming. In *Neural Networks, 1993., IEEE International Conference on*, pages 881–888. IEEE, 1993.

[42] Kenneth E Kinnear. Fitness landscapes and difficulty in genetic programming. In *Evolutionary Computation, 1994. IEEE World Congress on Computational Intelligence., Proceedings of the First IEEE Conference on*, pages 142–147. IEEE, 1994.

[43] Josef Kittler, Mohamad Hatef, Robert PW Duin, and Jiri Matas. On combining classifiers. *IEEE transactions on pattern analysis and machine intelligence*, 20(3):226–239, 1998.

[44] John R Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA, 1992. ISBN 0-262-11170-5.

[45] John R Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Press, Cambridge, MA, USA, 1994. ISBN 0-262-11189-6.

[46] John R Koza. Human-competitive results produced by genetic programming. *Genetic Programming and Evolvable Machines*, 11(3-4): 251–284, 2010.

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

[48] Ludmila I Kuncheva. *Combining pattern classifiers: methods and algorithms*. John Wiley & Sons, 2004.

[49] William B Langdon. The evolution of size in variable length representations. In *Evolutionary Computation Proceedings, 1998. IEEE World Congress on Computational Intelligence., The 1998 IEEE International Conference on*, pages 633–638. IEEE, 1998.

[50] William B Langdon and Wolfgang Banzhaf. A simd interpreter for genetic programming on gpu graphics cards. In *European Conference on Genetic Programming*, pages 73–85. Springer, 2008.

[51] William B Langdon and JP Nordin. Seeding genetic programming populations. In *European Conference on Genetic Programming*, pages 304–315. Springer, 2000.

[52] William B Langdon and Riccardo Poli. *Foundations of Genetic Programming*. Springer Publishing Company, Incorporated, 1st edition, 2010. ISBN 3642076327, 9783642076329.

[53] William B Langdon and Riccardo Poli. *Foundations of genetic programming*. Springer Science & Business Media, 2013.

[54] Joel Lehman, Jeff Clune, Dusan Misevic, Christoph Adami, Julie Beaulieu, Peter J. Bentley, Samuel Bernard, Guillaume Beslon, David M. Bryson, Patryk Chrabaszcz, Nick Cheney, Antoine Cully, Stéphane Doncieux, Fred C. Dyer, Kai Olav Ellefsen, Robert Feldt, Stephan Fischer, Stephanie Forrest, Antoine Frénoy, Christian Gagné, Leni K. Le Goff, Laura M. Grabowski, Babak Hodjat, Frank Hutter, Laurent Keller, Carole Knibbe, Peter Krcah, Richard E. Lenski, Hod Lipson, Robert MacCurdy, Carlos Maestre, Risto Miikkulainen, Sara Mitri, David E. Moriarty, Jean-Baptiste Mouret, Anh Nguyen, Charles Ofria, Marc Parizeau, David P. Parsons, Robert T. Pennock,

William F. Punch, Thomas S. Ray, Marc Schoenauer, Eric Shulte, Karl Sims, Kenneth O. Stanley, François Taddei, Danesh Tarapore, Simon Thibault, Westley Weimer, Richard Watson, and Jason Yosinksi. The surprising creativity of digital evolution: A collection of anecdotes from the evolutionary computation and artificial life research communities. *CoRR*, abs/1803.03453, 2018. URL http://arxiv.org/abs/1803.03453.

[55] Hubert W Lilliefors. On the kolmogorov-smirnov test for normality with mean and variance unknown. *Journal of the American statistical Association*, 62(318):399–402, 1967.

[56] Jason D Lohn, Gregory S Hornby, and Derek S Linden. An evolved antenna for deployment on nasa's space technology 5 mission. In *Genetic Programming Theory and Practice II*, pages 301–315. Springer, 2005.

[57] Sean Luke and Lee Spector. A revised comparison of crossover and mutation in genetic programming. *Genetic Programming*, 98(208-213):55, 1998.

[58] Sean Luke et al. Genetic programming produced competitive soccer softbot teams for robocup97. *Genetic Programming*, 1998:214–222, 1998.

[59] Penousal Machado, Juan Romero, Amílcar Cardoso, and Antonino Santos. Partially interactive evolutionary artists. *New Generation Computing*, 23(2):143–155, 2005.

[60] Henry B Mann and Donald R Whitney. On a test of whether one of two random variables is stochastically larger than the other. *The annals of mathematical statistics*, pages 50–60, 1947.

[61] Kenneth A Marko and Richard J Hampo. Application of genetic programming to control of vehicle systems. In *Intelligent Vehicles' 92 Symposium., Proceedings of the*, pages 191–195. IEEE, 1992.

[62] Atiya Masood, Gang Chen, Yi Mei, and Mengjie Zhang. Reference point adaption method for genetic programming hyper-heuristic in many-objective job shop scheduling. In *European Conference on Evolutionary Computation in Combinatorial Optimization*, pages 116–131. Springer, 2018.

[63] K Matsui. New selection method to improve the population diversity in genetic algorithms. In *Systems, Man, and Cybernetics, 1999. IEEE SMC'99 Conference Proceedings. 1999 IEEE International Conference on*, volume 1, pages 625–630. IEEE, 1999.

[64] Warren S McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, 1943.

[65] Yi Mei and Mengjie Zhang. Genetic programming hyper-heuristic for stochastic team orienteering problem with time windows. In *2018 IEEE Congress on Evolutionary Computation (CEC)*, pages 1–8. IEEE, 2018.

[66] Juan J Merelo, Pedro Castillo, Israel Blancas, Gustavo Romero, Pablo García-Sanchez, Antonio Fernández-Ares, Víctor Rivas, and Mario García-Valdez. Benchmarking languages for evolutionary algorithms. In *European Conference on the Applications of Evolutionary Computation*, pages 27–41. Springer, 2016.

[67] Brad L Miller and David E Goldberg. Genetic algorithms, selection schemes, and the varying effects of noise. *Evolutionary computation*, 4 (2):113–131, 1996.

[68] Brad L Miller, David E Goldberg, et al. Genetic algorithms, tournament selection, and the effects of noise. *Complex systems*, 9(3):193–212, 1995.

[69] Melanie Mitchell, Stephanie Forrest, and John H Holland. The royal road for genetic algorithms: Fitness landscapes and ga performance. In *Proceedings of the first european conference on artificial life*, pages 245–254, 1992.

[70] Thomas M Mitchell. *Machine Learning*. McGraw-Hill, Inc., New York, NY, USA, 1 edition, 1997. ISBN 0070428077, 9780070428072.

[71] David J Montana. Strongly typed genetic programming. *Evolutionary computation*, 3(2):199–230, 1995.

[72] Alberto Moraglio, Krzysztof Krawiec, and Colin G Johnson. Geometric semantic genetic programming. In *International Conference on Parallel Problem Solving from Nature*, pages 21–31. Springer, 2012.

[73] Kevin P Murphy. *Machine Learning: A Probabilistic Perspective*. The MIT Press, 2012. ISBN 0262018020, 9780262018029.

[74] Marco S Nobile and Giancarlo Mauri. Evolutionary inference of biochemical reaction networks accelerated on graphics processing units. In *High Performance Computing and Simulation (HPCS), 2013 International Conference on*, pages 668–670. IEEE, 2013.

[75] Michael O'Neill and Conor Ryan. Grammatical evolution. *IEEE Transactions on Evolutionary Computation*, 5(4):349–358, 2001.

[76] Michael O'Neill, Leonardo Vanneschi, Steven Gustafson, and Wolfgang Banzhaf. Open issues in genetic programming. *Genetic Programming and Evolvable Machines*, 11(3-4):339–363, 2010.

[77] Gisele L Pappa, Gabriela Ochoa, Matthew R Hyde, Alex A Freitas, John Woodward, and Jerry Swan. Contrasting meta-learning and hyper-heuristic research: the role of evolutionary algorithms. *Genetic Programming and Evolvable Machines*, 15(1):3–35, 2014.

[78] John Park, Yi Mei, Su Nguyen, Gang Chen, and Mengjie Zhang. An investigation of ensemble combination schemes for genetic programming based hyper-heuristic approaches to dynamic job shop scheduling. *Applied Soft Computing*, 63:72–86, 2018.

[79] Roger D Peng, Francesca Dominici, and Scott L Zeger. Reproducible epidemiologic research. *American journal of epidemiology*, 163(9): 783–789, 2006.

[80] Timothy Perkis. Stack-based genetic programming. In *Evolutionary Computation, 1994. IEEE World Congress on Computational Intelligence., Proceedings of the First IEEE Conference on*, pages 148–153. IEEE, 1994.

[81] Nelishia Pillay and Wolfgang Banzhaf. A genetic programming approach to the generation of hyper-heuristics for the uncapacitated examination timetabling problem. In *Portuguese Conference on Artificial Intelligence*, pages 223–234. Springer, 2007.

[82] Steven Pinker. *The language instinct: How the mind creates language*. Penguin UK, 2003.

[83] Riccardo Poli, William B Langdon, Nicholas F McPhee, and John R Koza. *A field guide to genetic programming*. Lulu. com, 2008.

[84] Alessandro Re, Mauro Castelli, and Leonardo Vanneschi. A comparison between representations for evolving images. In *International Conference on Evolutionary and Biologically Inspired Music and Art*, pages 163–185. Springer, 2016.

[85] Matteo Re and Giorgio Valentini. Ensemble methods: a review. 2011.

[86] Ingo Rechenberg. Evolutionsstrategie–optimierung technisher systeme nach prinzipien der biologischen evolution. 1973.

[87] Lior Rokach. Ensemble-based classifiers. *Artificial Intelligence Review*, 33(1-2):1–39, 2010.

[88] Juan J Romero. *The art of artificial evolution: A handbook on evolutionary art and music*. Springer Science & Business Media, 2008.

[89] Franz Rothlauf. Three elements of a theory of representations. In *Representations for Genetic and Evolutionary Algorithms*, pages 33–96. Springer, 2006.

[90] Arthur L Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of research and development*, 3(3):210–229, 1959.

[91] Robert E Schapire. The strength of weak learnability. *Mach. Learn.*, 5(2):197–227, July 1990. ISSN 0885-6125. doi: 10.1023/A:1022648800760. URL https://doi.org/10.1023/A:1022648800760.

[92] Amanda JC Sharkey. *Combining artificial neural nets: ensemble and modular multi-net systems*. Springer Science & Business Media, 2012.

[93] Lee Spector and Alan Robinson. Genetic programming and autoconstructive evolution with the push programming language. *Genetic Programming and Evolvable Machines*, 3(1):7–40, 2002.

[94] Lee Spector, Howard Barnum, Herbert J Bernstein, and N Swamy. Genetic programming for quantum computers. *Genetic Programming*, pages 365–373, 1998.

[95] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[96] Athanasios Tsanas and Angeliki Xifara. Accurate quantitative estimation of energy performance of residential buildings using statistical machine learning tools. *Energy and Buildings*, 49:560–567, 2012.

[97] Alan M Turing. Intelligent machinery. Report, 1948.

[98] Alan M Turing. Intelligent machinery, a heretical theory. *Philosophia Mathematica*, 4(3):256–260, 1996.

[99] Leonardo Vanneschi, Mauro Castelli, Luca Manzoni, and Sara Silva. A new implementation of geometric semantic gp and its application to problems in pharmacokinetics. In *European Conference on Genetic Programming*, pages 205–216. Springer, 2013.

[100] Leonardo Vanneschi, Mauro Castelli, Luca Manzoni, Krzysztof Krawiec, Alberto Moraglio, Sara Silva, and Ivo Gonçalves. Psxo: population-wide semantic crossover. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, pages 257–258. ACM, 2017.

[101] Kalyan Veeramachaneni, Ignacio Arnaldo, Owen Derby, and Una-May O'Reilly. Flexgp. *Journal of Grid Computing*, 13(3):391–407, 2015.

[102] David R White and Simon Poulding. A rigorous evaluation of crossover and mutation in genetic programming. In *European Conference on Genetic Programming*, pages 220–231. Springer, 2009.

[103] Frank Wilcoxon. Individual comparisons by ranking methods. *Biometrics bulletin*, 1(6):80–83, 1945.

[104] Garnett Wilson and Wolfgang Banzhaf. Deployment of parallel linear genetic programming using gpus on pc and video game console platforms. *Genetic Programming and Evolvable Machines*, 11(2):147–184, 2010.

[105] David H Wolpert. Stacked generalization. *Neural Networks*, 5:241–259, 1992.

[106] Daniel Yska, Yi Mei, and Mengjie Zhang. Genetic programming hyperheuristic with cooperative coevolution for dynamic flexible job shop scheduling. In *European Conference on Genetic Programming*, pages 306–321. Springer, 2018.

[107] Zhi-Hua Zhou. *Ensemble methods: foundations and algorithms*. CRC press, 2012.

[108] Zhi-Hua Zhou, Jianxin Wu, and Wei Tang. Ensembling neural networks: many could be better than all. *Artificial intelligence*, 137(1-2): 239–263, 2002.