CZECH TECHNICAL
UNIVERSITY
IN PRAGUE

F3

Faculty of Electrical Engineering
Department of Cybernetics

Ph.D. Thesis

Enhancements of Genetic Programming-based Symbolic Regression Algorithms

Ing. Jan Žegklitz
Ph.D. programme: Electrical Engineering and Information Technology (P2612)
Branch of study: Artificial Intelligence and Biocybernetics (3902V035)
zegkljan@fel.cvut.cz

July 2021
Supervisor: Ing. Petr Pošík, Ph.D.

Acknowledgement / Declaration

First and foremost, I would like to thank my supervisor, Petr Pošík, for his guidance and support. Whenever I thought I cannot see a way forward, he was ready to provide advice, to patiently discuss each problem, and eventually to help me continue. I express my deepest gratitude for everything he has done for me during this long PhD journey.

I would also like to express my thanks to Jiří Kubalík, who was one of my first teachers of evolutionary computation, who would then be the opponent of my master's thesis, and later my research colleague, and to Robert Babuška for taking me in for a research project which would spark one of the main topics of this thesis, and from whom I have learned a lot. I also thank Petr Olšák for creating and maintaining OpTEX and the template for this thesis, and for his immense help with typesetting.

My thanks also goes to a number of research grants and projects[1,2] that supported the research that gave birth to this thesis.

Last but definitely not least, I would like to thank all my friends and family for their very long, very patient, and very necessary support. This PhD journey would not be possible without any of them.

This thesis is submitted in partial fulfillment of the requirements for the degree of doctor of philosophy (Ph.D.). I hereby declare I have written this doctoral thesis independently and quoted all the sources of information used in accordance with methodological instructions on ethical principles for writing an academic thesis. Moreover, I state that this thesis has neither been submitted nor accepted for any other degree.

....

Jan Žegklitz
Prague, July 2021

[1] Czech Science Foundation grants No. 15-22731S and 17-01251S, Grant Agency of the Czech Technical University in Prague grants No. SGS14/194/OHK3/3T/13, SGS17/093/OHK3/1T/13, and SGS18/078/OHK3/1T/13.

[2] Access to computing and storage facilities owned by parties and projects contributing to the National Grid Infrastructure MetaCentrum, provided under the programme "Projects of Large Research, Development, and Innovations Infrastructures" (CESNET LM2015042), is greatly appreciated.

Abstrakt / Abstract

Symbolická regrese (SR) je druh regresní analýzy, kde je cílem najít model ve formě matematického výrazu, který je co nejjednodušší a, pokud možno, lidsky čitelný. Metody SR jsou většinou založeny na genetickém programování (GP).

V této práci se zabýváme vylepšením algoritmů SR založených na GP. Provádíme testovací studii, kde porovnáváme několik nedávných algoritmů SR na jednotné sadě testovacích problémů. Výsledky naznačují, že metody SR nejsou lepší než klasické metody strojového učení, ale skýtají prostor pro zlepšení.

Navrhujeme nový typ uzlu pro algoritmy SR založené na GP, který umožňuje zakódovat afinní transformace prostoru příznaků. Výsledky ukazují, že tyto uzly zlepšují výkon algoritmu.

Zavádíme problém dynamické SR, který je úzce spjat s oblastí posilovaného učení. Ukazujeme, že navržený nový typ uzlu je velmi přínosný pro tento druh problému, jelikož laditelné transformace umožňují rychle reagovat na změny.

V neposlední řadě se zaměřujeme na téma predikce fitness založené na podvzorkování trénovací datové sady. Navrhujeme modifikace dříve publikovaných metod, které značně zjednodušují algoritmus samotný a také zmenšují množství parametrů, které je třeba nastavit. Výsledky naznačují, že tato zjednodušení jsou vážnou alternativou k běžně užívanému koevolučnímu přístupu.

Klíčová slova: genetické programování; symbolická regrese; lineární regrese; strojové učení; konstrukce příznaků; fitness prediktory

Překlad titulu: Vylepšení algoritmů pro symbolickou regresi založených na genetickém programování

Symbolic regression (SR) is a kind of regression task where the goal is to find a model in the form of a mathematical expression that is as small as possible and, preferably, human-readable. SR methods are mostly based on genetic programming (GP).

In this thesis we focus on improving the GP-based SR algorithms. We conduct a benchmarking study that compares a number of recent SR algorithms on a common set of benchmarks. The results indicate that the SR methods are not superior to the classic machine learining methods, but there is a room for improvement.

We propose a new type of node for GP-based SR algorithms which allows for encoding affine transformations of feature space. The results show that these nodes improve the performance of the algorithm.

We introduce the task of dynamic SR which is closely connected to reinforcement learning. We show the good applicability of the proposed new type of node for this kind of task, as the tunable transformations allow for fast target tracking.

Lastly, we focus on the topic of fitness prediction based on subsampling the training dataset. We propose modifications of previous methods which significantly simplify the algorithms and decrease the number of parameters that need to be set. The results indicate, that these simplifications are a viable alternative to the commonly used coevolutionary approach.

Keywords: genetic programming; symbolic regression; linear regression; machine learning; feature construction; fitness predictors

Contents /

1 Introduction **1**
1.1 Problems of symbolic Regression by genetic programming . . . 2
1.2 Contributions of this thesis . . . 2
1.3 Thesis structure . . . 3

2 Preliminaries **4**
2.1 Genetic Programming . . . 4
2.1.1 Tree encoding . . . 5
2.1.2 Initialization . . . 5
2.1.3 Evaluation . . . 6
2.1.4 Elitism . . . 6
2.1.5 Selection . . . 6
2.1.6 Crossover . . . 6
2.1.7 Mutation . . . 7
2.2 Symbolic regression . . . 7

3 Overview of state-of-the-art symbolic regression algorithms **9**
3.1 Overview of selected symbolic regression algorithms . . . 9
3.1.1 Scaled Symbolic Regression . 9
3.1.2 Multi-Gene Genetic Programming . . . 10
3.1.3 Multiple-Regression Genetic Programming . . . 10
3.1.4 Evolutionary Feature Synthesis . . . 11
3.1.5 Fast Function Extraction . 11
3.1.6 Geometric Semantic Genetic Programming . . . 11
3.1.7 Geometric Semantic Genetic Programming with Reduced trees . . . 12
3.1.8 Eureqa . . . 12
3.1.9 ParetoGP . . . 12
3.1.10 Prioritized Grammar Enumeration . . . 13
3.1.11 Kaizen Programming . . . 13
3.1.12 Feature Engineering Automation Tool . . . 13
3.1.13 AI Feynman . . . 13
3.1.14 Deep Symbolic Regression . 14
3.2 Conceptual comparison . . . 14
3.2.1 Handling model complexity . . . 14
3.3 Summary . . . 15

4 Benchmarking state-of-the-art symbolic regression algorithms **16**
4.1 Selected algorithms . . . 16
4.2 Research questions . . . 18
4.3 Benchmarks and testing . . . 18
4.3.1 Artificial benchmarks . . . 19
4.3.2 Real-world benchmarks . . 20
4.3.3 Baseline algorithms . . . 20
4.3.4 Settings and usage of the algorithms . . . 21
4.3.5 Model complexity constraints . . . 24
4.3.6 Testing environment . . . 24
4.3.7 Testing methodology . . . 25
4.4 Results and discussion . . . 25
4.4.1 Results . . . 25
4.4.2 Discussion on error and complexity . . . 27
4.4.3 Global trends . . . 34
4.4.4 Comparison with previous research . . . 35
4.4.5 Discussion on running time . . . 36
4.5 Comparison with results from omitted algorithms . . . 36
4.6 Answers to research questions . 38
4.7 Summary and conclusions . . . 39

5 Feature space transformations for genetic programming **40**
5.1 Linear Combinations of Features . . . 40
5.1.1 Initializing the LCF weights . . . 42
5.1.2 Tuning the weights . . . 42
5.1.3 Operation modes of LCFs . 43
5.2 Related work . . . 44
5.3 LCFs in MGGP . . . 44
5.3.1 Genetic operators . . . 45
5.3.2 Incorporating LCFs in other SR algorithms . . . 46

5.4 Research questions 46
5.5 Experimental evaluation 46
5.5.1 Algorithm configurations . 47
5.5.2 Algorithm parameters . . . 47
5.5.3 Testing methodology and environment 48
5.5.4 Toy problems 49
5.5.5 Results on toy problems . . 49
5.5.6 Realistic problems 51
5.5.7 Results on realistic problems 52
5.6 Answers to research questions . 63
5.7 Summary and conclusions . . . 63

6 Feature space transformations in dynamic scenarios 65

6.1 Evolutionary Dynamic optimization 65
6.2 Dynamic symbolic regression . 66
6.3 Dynamic scenarios with gradual changes 67
6.4 Proof-of-concept experiment . . 69
6.5 Research questions 71
6.6 Experimental evaluation 71
6.6.1 Algorithm configuration . . 71
6.6.2 Benchmarks 72
6.6.3 Experiment settings 74
6.6.4 Results 74
6.6.5 Discussion 75
6.6.6 Intra-stage behaviour . . . 80
6.7 Answers to research questions . 81
6.8 Summary and conclusions . . . 82

7 Case-based fitness prediction 83

7.1 Related work 83
7.2 Extending the adaptive-size predictor approach 84
7.2.1 Uniform random sampling . 84
7.2.2 Local search for predictors 85
7.3 Research questions 86
7.4 Experimental evaluation 86
7.4.1 Base algorithms 86
7.4.2 Fitness prediction methods 87
7.4.3 Benchmarks 87
7.4.4 Performance metrics 88
7.4.5 Algorithm parameters . . . 90
7.4.6 Testing methodology 91
7.4.7 Results and discussion . . . 92
7.5 Experiments simulating more time-demanding evaluation 98
7.5.1 Results and discussion . . . 99
7.6 Answers to research questions 105
7.7 Summary and conclusion . . . 106

8 Conclusion 107

8.1 Future work 108

References 109

A List of abbreviations 119

B RL benchmark data generation 121

C Detailed performance plots of fitness prediction approaches 122

C.1 Regular experiments 122
C.2 Experiments with simulated time-demanding evaluation 129

D Author's publications 136

D.1 Publications related to the thesis topic 136
D.1.1 Journal publications . . . 136
D.1.2 Conference and workshop publications 136
D.1.3 Preprints and extended versions 137
D.2 Publications not related to the thesis topic 137
D.2.1 Journal publications . . . 137

E Curriculum Vitae 138

Tables / Figures

3.1 Conceptual comparison of algorithms. 15
4.1 Overall comparison of tested algorithms. 18
4.2 Definitions of artificial benchmarks 19
4.3 Training and testing sampling definitions 19
4.4 Real-world benchmarks summary 20
4.5 Function sets of individual algorithms 22
4.6 Sources of implementations of algorithms 24
4.7 Median test-RMSE 26
4.8 Statistical ranking of RMSEs .. 26
4.9 Median number of nodes 27
4.10 Statistical ranking of number of nodes 27
4.11 Median running times per dataset 36
4.12 Rough comparison of FEAT to our experiments 38
5.1 Algorithm configuration codenames 47
5.2 Algorithm parameters 48
5.3 Results on S5D toy problem 50
5.4 Results on RS5D toy problem . 50
5.5 Summary results on realistic problems 53
5.6 Results on K11C problem 54
5.7 Results on UB5D problem 55
5.8 Results on ASN problem 56
5.9 Results on CCS problem 57
5.10 Results on ENC problem 58
5.11 Results on ENH problem 59
5.12 Results on PS problem 60
5.13 Results on PS-I problem 61
5.14 Results on MM problem 62
6.1 Summary of RL benchmarks .. 73
6.2 Parameter values of algorithms 75
6.3 Statistical comparison of MGGP + LCF to all other algorithms 79
7.1 Datasets for fitness prediction . 87

2.1 Example of tree encoding of a program 5
2.2 Example of tree encoding of a mathematical expression 5
2.3 Illustration of subtree crossover .. 7
2.4 Illustration of subtree mutation .. 7
4.1 Complexity and performance on Koza-1 dataset 28
4.2 Complexity and performance on Korns-11 dataset 28
4.3 Complexity and performance on S1 dataset 29
4.4 Complexity and performance on S2 dataset 29
4.5 Complexity and performance on UB dataset 30
4.6 Complexity and performance on ENC dataset 30
4.7 Complexity and performance on ENH dataset 30
4.8 Complexity and performance on CCS dataset 31
4.9 Complexity and performance on ASN dataset 31
5.1 Comparison of model structure with and without LCF nodes 40
5.2 MGGP individual without and with LCFs 45
5.3 PS and PS-I datasets 52
5.4 MM dataset 52
5.5 Performance plot on K11C problem 54
5.6 Performance plot on UB5D problem 55
5.7 Performance plot on ASN problem 56
5.8 Performance plot on CCS problem 57
5.9 Performance plot on ENC problem 58
5.10 Performance plot on ENH problem 59
5.11 Performance plot on PS problem 60

7.2 Base algorithms' parameters... 90
7.3 Fitness prediction parameter grid points ... 91
7.4 Selected configurations ... 92
7.5 Aggregated results – EoR-R^2 .. 93
7.6 Aggregated results – RAUC ... 94
7.7 Aggregated results – RAUC ... 95
7.8 Aggregated results of experiments with simulated time-demanding evaluation – EoR R^2 ... 100
7.9 Aggregated results of experiments with simulated time-demanding evaluation – RAUC ... 101
7.10 Total number of generations. . 104

5.12 Performance plot on PS-I problem ... 61
5.13 Performance plot on MM problem ... 62
6.1 Example stages of two benchmarks ... 68
6.2 Dissimilarity of neighbouring stages ... 69
6.3 2-dimensional illustration of proof-of-concept dataset ... 70
6.4 Last-in-stage performance plot on artificial benchmark ... 76
6.5 Last-in-stage performance plot on PS benchmark ... 76
6.6 Last-in-stage performance plot on PS2 benchmark ... 77
6.7 Last-in-stage performance plot on MM benchmark ... 77
6.8 Last-in-stage performance plot on MM2 benchmark ... 78
6.9 Dissimilarity to the first stage . 80
6.10 Intra-stage behaviour on artificial benchmark ... 80
6.11 Intra-stage behaviour on PS benchmark ... 81
7.1 Illustration of performance metrics ... 89
7.2 Illustration of edge cases for the RAUC metric ... 89
7.3 Timing of algorithms on ASN dataset ... 96
7.4 Timing of algorithms on ParkinsonMotor dataset ... 96
7.5 Timing of algorithms on ParkinsonTotal dataset ... 97
7.6 Timing of algorithms on puma8NH dataset ... 97
7.7 Timing of algorithms on SupCon dataset ... 97
7.8 Timing of algorithms on WEC-A dataset ... 98
7.9 Timing of algorithms on ASN dataset with simulated time-demanding evaluation ... 102
7.10 Timing of algorithms on ParkinsonMotor dataset with

simulated time-demanding evaluation 102
7.11 Timing of algorithms on ParkinsonTotal dataset with simulated time-demanding evaluation 102
7.12 Timing of algorithms on puma8NH dataset with simulated time-demanding evaluation 103
7.13 Timing of algorithms on SupCon dataset with simulated time-demanding evaluation ... 103
7.14 Timing of algorithms on WEC-A dataset with simulated time-demanding evaluation 103
C.1 Performance on ASN dataset . 123
C.2 Performance on ParkinsonMotor dataset 124
C.3 Performance on ParkinsonTotal dataset 125
C.4 Performance on puma8NH dataset 126
C.5 Performance on SupCon dataset 127
C.6 Performance on WEC-A dataset 128
C.7 Performance on ASN dataset with simulated time-demanding evaluation ... 130
C.8 Performance on ParkinsonMotor dataset with simulated time-demanding evaluation 131
C.9 Performance on ParkinsonTotal dataset with simulated time-demanding evaluation ... 132
C.10 Performance on puma8NH dataset with simulated time-demanding evaluation 133
C.11 Performance on SupCon dataset with simulated time-demanding evaluation ... 134

C.12 Performance on WEC-A dataset with simulated time-demanding evaluation . . . 135

Chapter 1
Introduction

Evolutionary computation is a subfield of computer science which focuses on computational methods that are inspired by the natural phenomenon of evolution. In nature, living organisms compete for limited resources and try to survive and reproduce. As they reproduce, changes to their bodies and behaviours are inevitably introduced by random mutations when their DNA is copied. Some of these changes have no impact at all, some make the organisms better, and some make the organisms worse in the task of survival and reproduction. Those organisms that are better in this task are more likely to get a chance to reproduce while those that are not as good do not get this chance. Because of this, the successful organisms survive and reproduce while the unsuccessful ones eventually go extinct. It could be said that evolution optimizes the organisms' ability to prevail in the environment. An important aspect of evolution is that the optimal "solution" to the task of survival and reproduction is unknown (if one even exists), the optimization is achieved only by the random mutations and the non-random effect that those organisms that are not successful do not stay around to reproduce while the successful ones do. This phenomenon is also called the *survival of the fittest*.

Evolutionary algorithms (EAs) mimic the principle of natural evolution to perform optimization of difficult tasks. Instead of a population of organisms there is a population of candidate solutions (or just individuals) to the task, instead of the DNA there is an encoding of the individuals, instead of random mutations there are random variation operators (some of them called mutations), and, finally, instead of the implicit chance of reproduction there is often an explicit measure of *fitness*, usually called *fitness function*, that describes how good an individual is at solving the task, and an algorithm of selecting the individuals for reproduction based on this fitness. Similarly to natural evolution, EAs do not know how to find the optimal (or at least good enough) solution. But thanks to the selection pressure, the population is gradually driven towards better and better solutions.

An example of one type of EAs are genetic algorithms (GAs) [1]. GAs have a population of individuals of fixed size, and the individuals are encoded as binary strings of fixed length. The variation operators are crossover, where a portion of the binary strings is swapped between two individuals, and mutation, which randomly flips bits in the binary strings. There are several selection mechanisms, one of the most commonly used is a so-called tournament selection [2–3] where two (or more) individuals are picked from the population completely randomly and the one with the best fitness is selected.

Genetic programming (GP), first introduced by John Koza in [4], is very similar to GA – it also has a population, selection and variation operators. However, a major difference is that GP evolves tree structures (which represent computer programs) that are variable in size and are structured. This also comes with different variation operators: the crossover exchanges subtrees in the two individuals rather than a portion of a bit string, and the mutation replaces a subtree with randomly generated one instead of flipping a bit in a bit string.

Symbolic regression (SR) is a machine learning task, where the goal is to find a model that captures patterns in the given data well, and is in the form of an analytic mathematical function, that is as small and as simple as possible, ideally interpretable by a human. SR is a traditional application domain of GP [4], as the tree encoding used in GP is very natural to represent mathematical expressions.

Both GP and SR are described in more detail in Chapter 2.

1.1 Problems of symbolic Regression by genetic programming

While SR by GP has recorded some success, there are problems that limit the applicability in real life. The symptom of these problems is simply that GP is often not able to produce satisfactory SR models in a reasonable amount of time. We think there are two main reasons for this: searching for structure and parameters at the same time, and blindness of the search.

Simultaneous search for structure and parameters. SR algorithms based on GP are unique compared to conventional ML methods in that they are able to search for the structure of the model as well as for its parameters, both at the same time. This is usually viewed as a benefit over the conventional methods, because it frees the user from deciding on the structure of the model in advance [1], but it is a double-edged sword.

Searching for both the structure and its parameters at the same time can be inefficient. A candidate model structure can be discarded due to its bad performance but this performance might well be caused only by not having well-tuned parameters. The same effect, from the “other” side, is that a not particularly good structure gets chosen over a better one only because it happened to have better tuned parameters.

Blindness of search. GP, and EAs in general, are well suited for problems where there is little to no domain knowledge, i.e. so-called *black box* problems. However, SR is far from a black box problem.

In SR, the problem is very transparent: the training data (i.e. the features and the corresponding target values) is available by definition (otherwise there would be nothing to train on), and the error metric is known and often even designed by the user. However, standard GP does not utilize this information at all. The only mechanism of finding better models is to perform random modifications and then select them based on their errors. In other words, there is only the selection pressure that pushes towards better model.

1.2 Contributions of this thesis

Comparative study of algorithms. The first contribution of this thesis is a comparative study of several state-of-the-art algorithms on a common set of benchmarks. Before we conducted our comparative study [P2, J1], these algorithms have never been tested and compared. This work shines some light on how they compare to each other and allows the reader to make an informed decision as to what algorithm to choose, should there be such need.

[1] The user, however, still has to choose the building blocks and usually also the maximum size or complexity of the model.

Linear combinations of features. The second and main contribution of the thesis is a new type of node for GP – the Linear Combination of Features (LCF). This contribution addresses both problems mentioned in previous section. LCF nodes de facto construct new features as a linear combination of original features, exposing the coefficients of this combination as tunable parameters. These parameters are (or can be) tuned in an informed way by utilizing the error gradient, making the search not blind any more.

Dynamic symbolic regression. We introduced a dynamic SR task where the target values change over time. This task is heavily inspired by the process of learning value functions in reinforcement learning. This task differs from other dynamic SR tasks in the literature in that the changes are gradual and there are many more stages. We show that LCFs are a very good mechanism for this kind of problem.

Simplification of adaptive-size fitness predictors. We show that the usual way of using coevolution to find fitness predictors is not necessarily needed and can be substituted with local search or even simple random sampling, simplifying the algorithm, and reducing the number of parameters that need to be set or tuned.

1.3 Thesis structure

After this introductory chapter, we cover the basics of GP and SR (Chapter 2), if the reader is not familiar with this topic. We then proceed with an overview of the state-of-the-art algorithms for SR (Chapter 3). Chapter 4 is dedicated to benchmarking some of these algorithms and selection of the one we then use for the rest of this work. The Linear Combinations of Features are introduced and examined in Chapter 5. In Chapter 6 we introduce the task of dynamic symbolic regression, stemming from the processes that can be found in Reinforcement Learning, and show the applicability and good performance of LCFs for these problems. Chapter 7 focuses on the topic of fitness prediction and provides significant simplification of the current algorithms both in terms of the algorithm complexity itself as well as in number of parameters required by the algorithms. Chapter 8 concludes the thesis and offers ideas for possible future work.

Chapter 2
Preliminaries

In this chapter we familiarize the reader with the basic concepts regarding genetic programming and symbolic regression. A reader who is already familiar with these topics can safely skip this chapter.

2.1 Genetic Programming

Genetic programming (GP) [4] is a problem-solving paradigm based on the principles of natural evolution. GP is used to find solutions in the form of computer programs (generally speaking), for problems that are too hard to solve by other means, or when there is only a very limited amount of information known about the problem at hand. All that is needed to run a GP algorithm is the encoding of the solution that allows modifying the structure, and the ability to determine the quality of a solution, or even just to be able to compare and choose the better of two solutions.

The general structure of a GP algorithm is depicted in Algorithm 2.1. We are then going to cover all the important parts in the following subsections.

```
1  procedure GeneticProgramming
2      P ← InitializePopulation()              // initialization (see Section 2.1.2)
3      until termination condition
4          Evaluate(P)                         // evaluation (see Section 2.1.3)
5          P′ ← Best(P, n_elites)              // elitism (see Section 2.1.4)
6          until |P′| < |P|
7              p_1 ← Select(P)                 // selection (see Section 2.1.5)
8              p_2 ← Select(P)                 // selection
9              if rnd() < Pr_crossover
10                 Crossover(p_1, p_2)         // crossover (see Section 2.1.6)
11             end
12             if rnd() < Pr_mutation
13                 Mutate(p_1)                 // mutation (see Section 2.1.7)
14             end
15             if rnd() < Pr_mutation
16                 Mutate(p_2)                 // mutation
17             end
18             P′ ← P′ ∪ {p_1, p_2}
19         end
20         P ← P′
21     end
22     return the best individual in P
23 end
```

Algorithm 2.1. Pseudocode of a general GP algorithm.

Note. Algorithm 2.1 shows a so-called *generational* scheme. That means that in one generation (the outermost loop), a whole population of offspring is created, which then replaces the parent population (line 20). In the other commonly used scheme, only one or two offspring are created in each generation, and they are immediately put back into the population using some replacement strategy. For the sake of simplicity, we are going to focus on the generational scheme only.

Note on terminology. In the rest of the thesis, we will refer to this (Koza's [4]) style of GP as to "vanilla GP". Such GP algorithm has individuals composed of one tree, the crossover operator is the subtree crossover (see below), and the mutation operator is the subtree mutation (see below).

2.1.1 Tree encoding

By far the most commonly used representation, or encoding, of solutions in GP algorithms are trees, i.e. the programs are encoded as tree structures. An example of an encoding of a computer program into a tree is in Figure 2.1, and of a mathematical expression, which can also be considered a kind of a computer program, is in Figure 2.2.

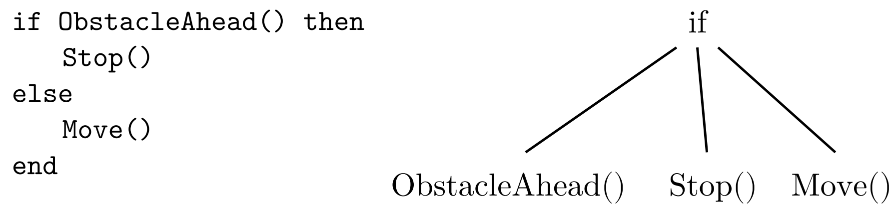


Figure 2.1. An example of how a simple program (left) can be represented by a tree structure (right).

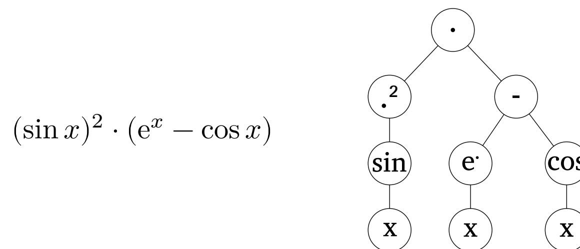


Figure 2.2. An example of how a simple mathematical expression (left) can be represented by a tree structure (right).

2.1.2 Initialization

In this step, the initial population of solutions is created. Usually, some kind of random initialization is used. One of the most commonly used procedures is Ramped Half'n'Half [4].

In this initialization procedure, the minimum and maximum depth are chosen beforehand. Then an equal number of individuals is generated for each depth between these limits. For each depth, half of the individuals is generated using the *grow* method, while the other half is generated using the *full* method. The two methods are very similar and differ only in which kind of nodes can be chosen at specified depth. A pseudocode of both methods combined (switched by the parameter $full$) is depicted in Algorithm 2.2. As can be seen, the only difference is whether a non-terminal (full), or either

```
1  function FullGrow(full, N, T, maxDepth)   // N – set of non-terminal nodes
2                                            // T – set of terminal nodes
3      if maxDepth = 1
4          return chooseRandom(T)
5      end
6      if full
7          node ← chooseRandom(N)
8      else
9          node ← chooseRandom(N ∪ T)
10     end
11     for i = 1, ···, numberOfChildren(n)
12         // set i-th child of node by recursive call with lower depth
13         child_i(node) ← FullGrow(full, N, T, maxDepth − 1)
14     end
15     return n
16 end
```

Algorithm 2.2. Pseudocode of the full and grow methods. If the method is called as $FullGrow(true, \cdots)$ it behaves as the *full* method. If it is called as $FullGrow(false, \cdots)$ it behaves as the *grow* method.

terminal or non-terminal (grow) node is allowed to be chosen at non-final depths. This means that the *full* method generates full, balanced trees up to the specified depth, while the *grow* method can generate less deep and unbalanced trees.

2.1.3 Evaluation

In this step, all individuals in the population are evaluated on the given problem, and their performance is recorded as their *fitness* value.

2.1.4 Elitism

It is common to select a number of best-performing individuals from the population and directly copy them into the offspring population. If this is done, there is no need to explicitly save the best-so-far individual, as the best is always present.

2.1.5 Selection

Selection represents the pressure on the individuals to get better. The purpose of the selection procedure is to select an individual from the parent population in such way that the better-performing ones are more likely to be chosen than the worse-performing. There are many selection procedures, but one of the most commonly used ones is the so-called *tournament selection*.

In tournament selection, a number[1] of individuals is chosen randomly from the parent population, and the one with the best fitness is selected.

2.1.6 Crossover

In crossover, "genetic" information is exchanged between the two parents. In GP, the standard type of crossover is so-called subtree crossover. In this type of crossover, a random point in each of the parent trees is chosen, and then the subtrees under this

[1] This number is then a parameter of the algorithm.

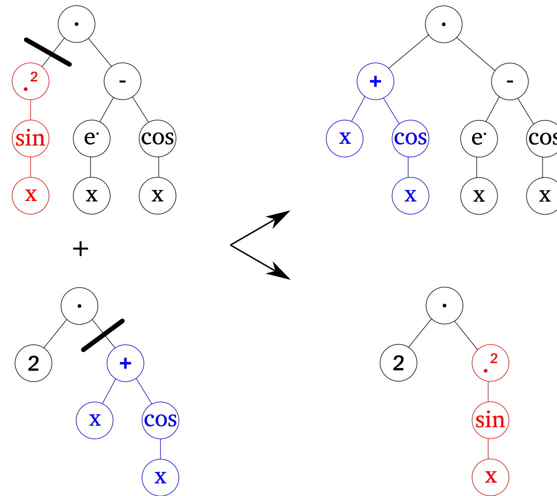


Figure 2.3. Illustration of a crossover event.

point are swapped between the parents, giving rise of two offspring. This is illustrated in the Figure 2.3.

Note that the crossover does not happen everytime, but is performed only with a predefined probability (see line 9 in Algorithm 2.1).

2.1.7 Mutation

The goal of mutation is to introduce new material into the population. In GP, the standard type of mutation is so-called subtree mutation. In this type of mutation, a random point is chosen in the individual, and then the subtree under this point is discarded, and a new randomly generated[2] subtree is placed there. A mutation event is illustrated in the Figure 2.4.

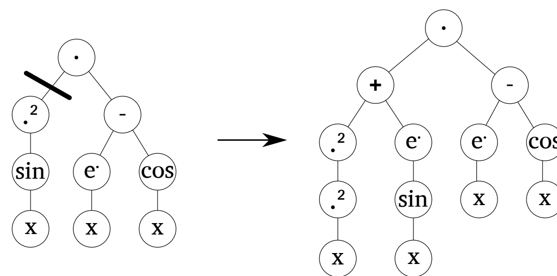


Figure 2.4. Illustration of a mutation event.

Similarly to crossover, mutation is also applied only with a certain predefined probability (see lines 12 and 15 in Algorithm 2.1).

2.2 Symbolic regression

Symbolic regression (SR) is an inductive learning task, where the goal is to find a model in the form of a (preferably simple) symbolic mathematical expression that fits the available training data.

[2] Typically either the *full* or *grow* method is used, with a predefined maximum depth.

Formally, given a set of data samples $\mathbf{X} = \{\mathbf{x}_i \mid i = 1, 2, \cdots, n; \mathbf{x}_i \in \mathbb{R}^N\}$, their corresponding target values $\mathbf{y} = \{y_i \mid i = 1, 2, \cdots, n; y_i \in \mathbb{R}\}$, and an error function e, find a function f such that $e(\hat{\mathbf{y}}, \mathbf{y})$ is minimized, where $\hat{\mathbf{y}} = \{\hat{y}_i | i = 1, 2, \cdots, n; \hat{y}_i = f(\mathbf{x}_i)\}$ is a set of estimated target values.

However, the user is usually not as much interested in the goodness of fit on the training data, but rather in how well the model actually captures the patterns present in the data, that is, how well the model *generalizes*. In practice, this is measured using a *testing* dataset, which is identically independently distributed, i.e. the testing data come from the same distribution as the training data, but are independent of it. A model that fits training data very well, but exhibits great error on testing data, is said to exhibit *overfitting*, or fails to generalize.

SR models are analytic functions, in principle suitable to interpretation and analysis. Traditionally, GP has been used as the engine for SR [4], or, in other words, SR has been one of the first applications of GP. The tree-based encoding used in GP is very suitable to build SR models as mathematical expressions translate to trees very naturally, as could be seen in the previous section.

For the task of SR, GP is usually slightly extended compared to the basic structure, which we outlined in the previous section, in order to allow for better handling of numerical constants. In principle, a GP algorithm can use just a single non-zero constant and derive other values by evolving arithmetic operations manipulating this constant. That is, however, inefficient, and therefore, other techniques are used. One common technique is called "ephemeral random constant", or ERC [4]. ERC is a special terminal symbol, which, when it is chosen in a tree generation procedure, produces a randomly generated[3] numerical constant, which is put in its place. Another common technique is the use of numeric mutations [5] – a second mutation operator beside subtree mutation[4] – where numeric leaf nodes are selected and their value is perturbed by a randomly generated[3] number.

While the conventional machine learning (ML) techniques usually fit models with a structure fixed in advance and only tune their parameters, GP searches a much broader class of possible models limited only by the available function and terminal symbols and maximal model complexity, i.e., GP searches also for a useful structure of the model. Such a system may reach impressive results [6–7] when given good data and enough time, sometimes even recovering the true equations describing the underlying phenomenon which generated the observed data.

[3] From a predefined distribution.

[4] Whether they can both be applied to a single individual, or they are mutually exclusive with stochastic choice between them, is up to the implementation of particular algorithm.

Chapter 3
Overview of state-of-the-art symbolic regression algorithms

In this short chapter, we provide an overview of the current state of SR algorithms. We present a number of algorithms which we consider important and/or state of the art. We also perform a high-level conceptual comparison of the algorithms to highlight the main features and differences.

3.1 Overview of selected symbolic regression algorithms

In this section we give a broad overview of important SR algorithms.

As will be seen in the following subsections, there is a notable trend among the algorithms, which is that the models produced by the algorithms has a form of linear combination of one or more (non-linear) functions:

$$f(\mathbf{x}) = a_0 + \sum_{i=1}^{k} a_k \cdot B_i(\mathbf{x}) \tag{3.1}$$

where a_i are real coefficients, B_i are (possibly non-linear) basis functions, and k is the number of basis functions in the model. This form of model could be called a Generalized Linear Model (GLM) [8]. However, the GLM framework is more general than that. For this reason, we shall call such a model, as defined in Equation (3.1), a **Linear Combination of Basis Functions**, or **LCBF**.

In this overview, we focus on algorithms that produce the model in the form of an LCBF, and on a selection of other algorithms that we consider important.

3.1.1 Scaled Symbolic Regression

Scaled Symbolic Regression (SSR) [9] is a simple extension of vanilla GP. In SSR, individuals are single trees, as in vanilla GP. However, before the tree is evaluated, the output of the tree is scaled and shifted, i.e. the tree expression becomes $f(x) \to a + bf(x)$, where a and b are parameters determined by running simple linear regression on the output of the original tree with respect to the training data. Therefore, the models in SSR are in the form of a LCBF (see Equation (3.1)) with $k = 1$.

This extension is motivated by the idea that a tree might have a good structure but it would be rendered unfit only because it does not have the right linear parameters. The benefit of this extension is significant – it improves the performance of the algorithm from the very start and it lifts the burden of finding the simple linear parts from the evolutionary process.

3.1.2 Multi-Gene Genetic Programming

Multi-Gene Genetic Programming, or MGGP [10–11], could be viewed as further extension of Scaled Symbolic Regression. In MGGP, an individual is composed of not one but multiple trees. These trees are linearly combined, forming an LCBF, with the coefficients being computed optimally with respect to the error on the training data using multiple linear regression. From the individual evaluation perspective, the difference from Scaled Symbolic Regression is only in the number of expression trees considered for linear regression.

With the different structure of an individual (a set of trees instead of a single tree), MGGP introduces a new crossover operator, called high-level crossover, which exchanges whole genes between two individuals, rather than subtrees of some of their genes. The original subtree crossover operator is kept, and the operator used for crossover is chosen stochastically from these two.

MGGP has been shown to be faster and more accurate than vanilla GP [10] and also a comparable or better alternative to classical methods like Support Vector Regression and Artificial Neural Networks [12].

3.1.3 Multiple-Regression Genetic Programming

Multiple-Regression Genetic Programming, or MRGP [13], is another extension of vanilla GP. The motivation behind MRGP is that in vanilla GP, the selection pressure is on the whole program (expression) rather than its building blocks, or subprograms (subexpressions). MRGP approaches this issue by changing how the model output is computed. In MGRP, the expression is broken down into a set of subexpressions of the original expression, and then the outputs of these subexpressions are linearly combined (hence producing a LCBF) using multiple linear regression. In order to deal with possible incompatibilities with linear regression[1], the authors employ Least Angle Regression.

The authors propose two variants – post-run MRGP and inline MRGP. The post-run variant is only an extra step run once on the final solution from another SR algorithm. The authors propose various strategies of which subexpressions to select for the linear combination: root-only (which collapses into a simple univariate regression, similar to what Scaled Symbolic Regression does with each individual), root and leaves, all subexpressions (including the root and leaves), root and variables (i.e. the features, whether or not they are present in the tree), and subexpressions and variables (if a variable is also present in the tree, it is not selected twice). The authors have shown that this extra step improves the performance of the base SR algorithm.

The inline variant uses the breakdown and linear combination on each individual during a SR algorithm run. In order to control for the size of the model, the authors employ a model complexity metric and use the NSGA-II algorithm [14] which uses the prediction error and the complexity metric as criteria to drive the evolution. The authors try several complexity metrics which are all of similar performance but the metric "sum-t" (sum of t-statistics, we refer the reader to the original paper [13] for details) produced the "most reliable" results, meaning being among the best performers and showing low variance between the individual runs. Overall, the inline variant produced even smaller errors than the post-run variant.

[1] Subexpressions can be linearly dependent, or the tree size (and therefore the number of subexpressions) is greater than the number of datapoints in the training set.

3.1.4 Evolutionary Feature Synthesis

In Evolutionary Feature Synthesis, or EFS [15], the population does not consist of complete models but rather of features, or bases, which, collectively, form a single model in the form of LCBF.

The initial population is formed by the original features of the dataset. Then, in each generation, a model is composed of the features in the current population by Pathwise Regularized Learning and is stored if it is the best. The next step in a generation is the composition of new features by applying unary and binary functions to the features already present in the current population. This way, more complex features are created from simpler ones. Also, the features are selected during this composition step according to the Pearson correlation coefficient with the feature's parents.

EFS does not build the symbolic model explicitly – it works with the data of the features in a vectorial fashion and only stores the structure for logging purposes. This is possible because the operators in EFS are only constructive, i.e. only whole expressions are combined into bigger ones, and therefore the subexpressions do not need to be reevaluated. This results in a very fast algorithm.

The original paper [15] reports EFS being comparable to neural networks and similar or better than MRGP which itself was reported to outperform vanilla GP, multiple regression, and SSR.

3.1.5 Fast Function Extraction

Fast Function Extraction, or FFX [16], is a *deterministic* algorithm for symbolic regression. It first exhaustively generates a massive set of basis functions, which are then linearly combined using Pathwise Regularized Learning [17–18] to produce sparse models in the form of LCBF. In this respect, FFX is similar to EFS: in EFS the features may be more complex (depending on the complexity constraints) and are generated stochastically while in FFX the individual features are relatively simple and are generated systematically and exhaustively but in both algorithms a set of features collectively forms a single LCBF model. The algorithm produces a Pareto-front of models with respect to their accuracy and complexity. It is up to the user to choose the final model.

There are two kinds of bases that are generated: univariate bases and bivariate bases. Univariate bases are

- a variable raised to a power (chosen from a fixed set of options), and
- (non-linear) functions applied to another univariate base.

Bivariate bases are products of all pairs of univariate bases excluding the pairs where both the bases are of function-type. The author of FFX argues that products of two function-type bases are "deemed to be too complex" (in [16], Section 4, paragraph "FFX Step One"). FFX also includes a trick that allows it to produce rational functions of the bases using the same learning procedure. When executed, the algorithm runs multiple times with various features of the algorithm turned on and off, combining the results of all these runs into a single final Pareto-front.

The original paper [16] reports FFX to be more accurate than many classic methods including vanilla GP, neural networks and SVM.

3.1.6 Geometric Semantic Genetic Programming

Geometric Semantic Genetic Programming, or GSGP [19], is a novel view of the SR problem. In GSGP, the focus of the evolution is the semantics (the output values of the candidate solutions) rather than the syntax (the actual trees and expressions). The

semantic space is an n-dimensional Euclidean space where n is the size of the training dataset. Each candidate function maps into this semantic space as a single point with coordinates equal to the output values the function produces for each individual datapoint. From this point of view, the goal is to find a function that is as close as possible to the point defined by the known target values.

GSGP uses simple linear operators to search the semantic space. Geometric Semantic Crossover takes two trees from the population and creates an offspring by making a (weighted) average of the two parents. Geometric Semantic Mutation takes an individual from the population and creates a linear combination of this individual and a tree composed of a difference of two randomly generated trees. The rest of the algorithm is very similar to vanilla GP – the individuals are selected, evaluated, and reproduced/modified with the two operators.

From the point of view of the geometric semantic operators, the fitness landscape is a cone which is easy to search, even using just the mutation operator. GSGP is able to converge quite quickly (compared to vanilla GP) and steadily. It is also resistant to overfitting due to the small steps it makes towards the optimum. The major disadvantage of GSGP is the fact that the size of the solutions grows exponentially (with the crossover operator) or linearly (with the mutation operator) over time which results in huge trees [20].

3.1.7 Geometric Semantic Genetic Programming with Reduced trees

Geometric Semantic Genetic Programming with Reduced Trees, or GSGP-Red [20], is a recent extension to GSGP [19] we just discussed. The algorithm is equivalent to standard GSGP in that it builds the models using exactly the same operators. GSGP-Red extends GSGP by adding a simplification step after the application of the geometric semantic crossover and mutation operators which considerably reduces the size of the individuals which would otherwise (i.e. in canonical GSGP) grow exponentially.

Although GSGP does not produce LCBFs by design, the models are linear combinations due to the nature of the geometric semantic operators. In pure GSGP, the final model is a linear combination of two or three trees (depending on whether the last applied operator was crossover or mutation). However, the individual is composed of recursive application of the linear combination introduced by the genetic operators. GSGP-Red's simplification procedure expands the linear combinations introduced by the genetic operators which produces a "flat" linear combination. Therefore, the final models are, in fact, LCBFs.

3.1.8 Eureqa

Eureqa [7, 6] is a commercial SR tool utilizing coevolution of fitness predictors [21], and Age-Fitness Pareto Optimization [22]. The coevolution of fitness predictors is employed in order to reduce the amount of time spent in evaluation and to focus on datapoints that are the most important ones. Age-Fitness Pareto optimization [22] introduces age of the solutions as a second objective in Pareto front optimization, i.e. the algorithm aims to optimize the solutions to have the best fitness with minimum age.

The tool is, however, closed-source and therefore cannot be properly analysed.

3.1.9 ParetoGP

ParetoGP [23–24] is a GP-based algorithm that considers the model complexity as one of the objectives and uses a Pareto-front archive of the best solutions (in a multi-

objective sense). The algorithm picks parents for crossover from this archive and from the population.

3.1.10 Prioritized Grammar Enumeration

Prioritized Grammar Enumeration, or PGE [25], is a *deterministic* SR algorithm that works with tree-based representation and uses Pareto non-dominance sorting. However, instead of stochastic operators used in GP-based algorithms, it uses grammar production rules to systematically search the space of mathematical formulas. PGE uses simplification procedures to reduce the size of the trees while keeping the expressions syntactically equivalent. Tree nodes for addition and multiplication are n-ary rather than binary and the algorithm imposes an ordering on the children of these operators, reducing the search space. The algorithm utilizes the Levenberg-Marquardt algorithm to optimize the numerical coefficients in the formulas. Lastly, the algorithm uses integer prefix tree to store and determine whether an expression has already been seen or not, evaluating each form only once, even when derived along different paths through the grammar production rules.

3.1.11 Kaizen Programming

Kaizen Programming, or KP [26], is an algorithm inspired by the "Kaizen" methodology. For the symbolic regression task, the algorithm builds models in the form of an LCBF. The algorithm cycles through four phases called *plan*, *do*, *check*, and *act*. In the *plan* phase, new basis functions are generated from the ones already available (at the beginning, only the original features are available), similarly to crossover and mutation in GP. In the *do* phase, outputs of all basis functions are computed. In the *check* phase, multiple linear regression is applied to the outputs computed in the previous phase, producing coefficients for each of the basis functions. These coefficients are then used to assess the contribution of their corresponding basis functions. In the *act* phase, the basis functions are filtered based on their contribution. The algorithm is essentially greedy and a restarting strategy is employed in order to prevent getting stuck in local optima.

3.1.12 Feature Engineering Automation Tool

Feature Engineering Automation Tool, or FEAT [27], is a novel algorithm on the borderline between GP and neural networks. Similarly to MGGP, the algorithm evolves a set of trees instead of a single one, which are then combined using ridge regression, forming an LCBF. Weights are assigned to inputs of each differentiable operator and are tuned using backpropagation and gradient descent. One of the main novelties of this algorithm is using feedback from the ridge regression: trees with smaller coefficients are considered more suitable to be targeted by variation operators. The algorithm uses ε-Lexicase selection [28] and the survival scheme of NSGA-II [14] using three objectives: model error, complexity, and disentanglement which aims to reduce correlations between the trees inside an individual. An extension of the algorithm [29] introduced semantic-aware crossover operators. These operators utilize the information of the output of individual trees inside each parent in order to minimize the residual of the offspring. The algorithm was shown to be competitive with conventional ML methods and producing orders of magnitude simpler models.

3.1.13 AI Feynman

AI Feynman [30] is a novel physics-inspired algorithm designed to recover exact formulas of physical phenomena. The fundamental idea of the algorithm is to exploit common

properties of equations describing physical phenomena to simplify the problem into one that can be solved easily. The simplification techniques and principles are: dimensional analysis using known units, translational, rotational and scaling symmetries, and additive and multiplicative separability. The algorithm applies these techniques and calls itself recursively.

The algorithm assumes smoothness of the sought function, i.e. that it is differentiable in its domain, and trains a fully connected 6-layer FFNN to make a good approximation of the sought function. This neural network is then used to discover symmetries and separability. The search engine used when the problem can no longer be simplified is a simple brute force search. The authors demonstrated the performance of the algorithm by letting it recover 100 equations extracted from Richard Feynman's *The Feynman Lectures on Physics* and 20 more physical equations that the authors deem famous and complex. The algorithm has been able to recover all of the 100 equations and 18 of the 20 extra equations.

3.1.14 Deep Symbolic Regression

Deep Symbolic Regression, or DSR [31], is very recent addition to the pool of SR techniques. DSR utilizes a recurrent neural network (RNN) to produce symbolic expression trees. The network produces a preorder traversal of the tree, one node at a time, by producing a probability distribution over the available nodes at each step and the node is then chosen with that distribution. The parent and sibling nodes for the currently produced node are passed as an input to the network. Reinforcement learning (RL) is used to train the RNN. Citing from [31], Section 3.2: “In this view, the distribution over mathematical expressions $p(\tau|\theta)$ is like a policy, sampled tokens are like actions, the parent and sibling inputs are like observations, sequences corresponding to expressions are like episodes, and the reward is a terminal, undiscounted reward only computed when an expression completes.” The authors propose and use a risk-seeking policy gradient to optimize the RNN.

3.2 Conceptual comparison

Table 3.1 provides a high-level conceptual comparison of the algorithms.

3.2.1 Handling model complexity

The discussed algorithms handle the issue of model complexity in different ways. GSGP does not handle model complexity at all. GSGP-Red also does not explicitly handle the complexity but effectively reduces it thanks to the simplification procedures employed. SSR is not fully an algorithm itself but rather an improvement of how to work with the individual which is applicable to a range of algorithms which have their own way of handling the complexity. MGGP is similar in this regard to SSR but there is a hard limit (which is a parameter of the algorithm) on the number of trees in an individual. EFS uses LASSO regression to regularize and limit the number of bases in the final model. Other than that, it does not limit the size of the bases as an algorithm but its implementation by the authors [32] has a hardcoded limit of depth of 5. FFX enumerates all possible bases allowed by the algorithm, then uses elastic net to select from them. The generation procedures result in maximum depth of 5. The algorithm also produces a non-dominated set of models of varying complexity and it is up to the user to select the final model. AI Feynman relies on the problem simplification/decomposition to transform it into a problem simple enough that it can be solved by brute

Algorithm	Family	Is model LCBF?	Individual	ref.
SSR	GP	single-base	tree	[9]
MRGP	GP	yes	tree	[13]
MGGP	GP	yes	set of trees	[10]
EFS	GP	yes	tree	[15]
FFX	enumeration (deterministic)	yes	no concept of individuals	[16]
GSGP	GP	no	tree	[19]
GSGP-Red	GP	yes	LCBF	[20]
Eureqa	GP, coevolution	no	tree	[7]
ParetoGP	GP	no	tree	[23–24]
PGE	dynamic programming (deterministic)	no	tree	[25]
KP	restarted stochastic greedy search	yes	tree	[26]
FEAT	GP, $\mu + \lambda$ evolution strategy	yes	LCBF	[27, 29]
AI Feynman	almost deterministic[1]	no	no concept of individuals	[30]
DSR	RNN/reinforcement learning	no	no concept of individuals	[31]

[1] Except for batch shuffling when training the FFNN.

Table 3.1. Conceptual comparison of the algorithms.

force search; otherwise, it fails. There is no explicit complexity handling mechanism although the algorithm is designed to produce *simple* expressions that can be encountered in the real world, especially in physics. PGE does not have explicit complexity handling mechanism either but, similarly to EFS, it starts with minimal expressions and only expands them, therefore covering smaller solutions first. FEAT uses multi-objective approach with the complexity as one of the objectives. Their complexity measure is defined recursively as the complexity of an expression is the sum of complexities of its subexpressions multiplied by a complexity weight of the operator at the root of the expression. In DSR, there is a hard limit (which is a parameter) on the minimum and maximum length of an expression.

3.3 Summary

In this chapter we have provided an overview of a number of SR algorithms that we consider state of the art and/or important for the field. As we have shown, e.g. in Table 3.1, majority of the algorithms produce models in form of LCBFs.

In the next chapter we focus on a selection of the algorithms that produce LCBF models. We subject them to testing on a common set of benchmarks and select one of them which will be used as the base algorithm for the rest of this thesis.

Chapter 4
Benchmarking state-of-the-art symbolic regression algorithms

In Chapter 1 we discussed two problems that GP-based SR algorithms exhibit: possible inefficiency of searching for both structure and parameters at the same time, and the blindness of search while the SR problem is not a black box problem. In previous chapter we outlined a number of SR algorithms which employ Linear Regression (LR) in order to (partially) tackle both problems. Models found by these algorithms exhibit relatively small error right from the start of the algorithm and are claimed to be orders of magnitude faster than SR algorithms based on vanilla GP.

A systematic comparison of such algorithms on a set of common problems, however, has been missing. In this chapter we conceptually and experimentally compare some of these algorithms. The goal is to make a comparison on a common set of benchmarks, and, based on the results, select one algorithm we are going to use as the basis for further improvements in the rest of this work.

This chapter is based on and slightly extends the research done in [P2, J1].

4.1 Selected algorithms

From the algorithms presented in Chapter 3, we have selected the following:

1. GPTIPS [33], an implementation of MGGP,
2. FFX [16], an example of a non-evolutionary deterministic method,
3. EFS [15], a recent evolutionary method for fast creation of interpretable SR models, and
4. GSGP-Red [20], an extended version of GSGP which employs a simplification procedure that reduces the solution size.

These methods were reported by their authors as successful SR methods.

We have selected these algorithms for the following reasons:

- they all have available open-source implementations that allow us to perform proper testing (Eureqa and ParetoGP are not openly available),
- we consider them a good representation of what seems to be the general trend of producing models in the form of LCBFs (GSGP, Eureqa, ParetoGP, PGE, AI Feynman, and DSR do not produce LCBFs),
- some of the algorithms we mentioned in previous chapter were not yet published when this research was performed (namely FEAT, AI Feynman, and DSR), so they obviously could not be included in the comparison, and
- some of the algorithms are ancestors of the ones included in the comparison (SSR $\rightarrow$ MGGP, MRGP $\rightarrow$ EFS, GSGP $\rightarrow$ GSGP-Red)

```
1  procedure RandomSearch
2      I* ← GenerateIndividual()
3      until termination condition
4          I ← GenerateIndividual()
5          if f(I) > f(I*) then                     // evaluate and compare
6              I* ← I                               // store if best so far
7          end
8      end
9      output I*
10 end
11 function GenerateIndividual
12     G ← ∅
13     n_g ← U(1, n_max)                            // choose # of genes
14     repeat n_g-times                             // generate each gene
15         depth ← U(depth_min, depth_max)          // choose depth
16         method ← R({full, grow})                 // choose full or grow method
17         gene ← method(depth)
18         G ← G ∪ gene
19     end
20     return G
21 end
```

Algorithm 4.1. Pseudocode for the RdS algorithm, where f is the fitness function (which includes the linear regression to determine the intercept and genes' coefficients), $\mathcal{U}(a, b)$ represents an integer between and including a and b chosen uniformly randomly, and $\mathcal{R}(\mathbb{S})$ represents an element from the set $\mathbb{S}$ chosen uniformly randomly.

In addition to the four selected algorithms, we also include a **Random Search (RdS)**. It is a trivial algorithm (see Algorithm 4.1) we designed for the purpose of providing a baseline of a minimum that can be reasonably expected from linear combinations of multiple (non-linear) expressions. The algorithm is intentionally very simple – it has no population and only performs a random search, storing the best solution found so far over the course of a run. The algorithm has a single MGGP-style individual (i.e. in the same form as GPTIPS has) which is randomly generated in each iteration and stored if it is better than the best one so far. As in MGGP/GPTIPS, the individual is a set of one or more trees which are combined with a linear combination with coefficients determined using classic LR on the outputs of the trees, exactly as MGGP/GPTIPS does. The individual generation procedure is inspired by the Ramped Half-n-Half population initialization method [4]: first a number of genes is uniformly randomly picked from the allowed range, then for each gene the maximum depth of that gene is uniformly randomly picked from the allowed range, then the *full* or *grow* method is picked randomly, and, finally, the gene in question is generated using the picked method and up to the picked depth. As a part of the evaluation, the coefficients of the top-level linear combination are determined in the same way as in GPTIPS.

A high-level comparison of the selected algorithms is provided in Table 4.1.

algorithm	LR technique	type of alg.	individuals	raw alg. output
GPTIPS	pure LR	evolutionary	complete models	the last population of models
EFS	LASSO	evolutionary	bases of a bigger model	single best-performing model
FFX	elastic net	deterministic	no concept of individuals	Pareto-front of models
GSGP-Red	none	evolutionary	complete models	single best-performing model
Random Search	pure LR	random	complete model	single best-performing model

Table 4.1. Overall comparison of the tested algorithms.

4.2 Research questions

Now, when we selected the algorithms of interest, we ask the three research questions.

RQ1: What are the performance differences between the algorithms? We do not really expect that one of the algorithms would produce better models than the others in all reasonable circumstances. We are more interested in the types of differences we can expect from these algorithms when applied to the same regression problems.

RQ2: How do the algorithms compare to reasonable baselines? We propose two types of baseline:

- a trivial algorithm that uses random search to generate bases for LCBF which are then fitted using LR, and
- conventional machine learning (ML) techniques.

The random search based algorithm should show what can be expected from LCBFs at minimum, i.e. the importance of evolution in the algorithms. Comparison with conventional ML techniques should show how competitive the SR algorithms are.

RQ3: What algorithm should we choose for further experimentation? We want to choose an algorithm that has good performance, but the algorithm design also plays a major role as it can significantly restrict (or support) extensibility.

4.3 Benchmarks and testing

For testing, we have selected five artificial and four real-world benchmarks. The artificial benchmarks cover various types of complexities and features. An important feature of

all the artificial benchmarks (except for Koza-1) is that they contain internal constants, which is challenging for all the algorithms. The quality of the results is judged just by the testing error: we shall thus see whether the limited ability to learn the internal constants is a show-stopper for these algorithms.

4.3.1 Artificial benchmarks

All the datasets except for UB were picked from the comprehensive summary of benchmarks in [34]. Table 4.2 presents a summary of the used artificial benchmarks: their definitions, number of dimensions and their original source. Table 4.3 presents the training and testing sampling of those datasets. Using the notation from [34]:

- the expression $U[a, b, c]$ means c random samples from uniform distribution on the interval $[a, b]$ for each variable;
- the expression $E[a, b, c]$ means a grid in the interval $[a, b]$ with spacing of c (including the boundaries) for each variable.

Name	Definition	# of features	Ref
Koza-1	$f_1(x) = x^4 + x^3 + x^2 + x$	1	[4]
Korns-11	$f_2(x_1, x_2, x_3, x_4, x_5) = 6.87 + 11 \cos(7.23x_1^3)$	5	[35]
S1	$f_3(x) = \mathrm{e}^{-x} x^3 \sin(x) \cos(x)(\sin^2(x) \cos(x) - 1)$	1	[36]
S2	$f_4(x_1, x_2) = (x_2 - 5) f_3(x_1)$	2	[36]
UB	$f_5(x_1, x_2, x_3, x_4, x_5) = \frac{10}{5+\sum_{i=1}^{5}(x_i-3)^2}$	5	[36]

Table 4.2. Definitions of the artificial benchmarks.

Name	Training sampling	Testing sampling
Koza-1	$U[-1, 1, 20]$	$U[-1, 1, 100]$
Korns-11	$U[-50, 10, 10000]$	$U[-50, 10, 10000]$
S1	$E[-0.5, 10.5, 0.1]$	$E[-0.5, 10.5, 0.05]$
S2	$x_1 = E[-0.5, 10.5, 0.1]$	$x_1 = E[-0.5, 10.5, 0.05]$
	$x_2 = E[-0.5, 10.5, 2]$	$x_2 = E[-0.5, 10.5, 0.5]$
UB	$U[-0.25, 6.35, 1024]$	$U[-0.25, 6.35, 5000]$

Table 4.3. Description of the training and testing sampling. Note that each variable in S2 has its own sampling type.

Koza-1 [4] is a classic, easy-to-solve SR benchmark. It shall test the ability of the algorithms to fit a very simple function. Although [34] advises not to use these easy benchmarks, we included it as a sanity check.

Korns-11 [35] is specific in the fact that the output depends on only one of the 5 input features and also by the presence of internal constants. The function is hard to fit because of the high frequency components.

Salustowicz 1D (S1) [36] (called Vladislavleva-2 in [34]) is defined by a single, rather complex term. It does not fit the LCBF structure well.

Salustowicz 2D (S2) [36] (called Vladislavleva-3 in [34]) has similar features as S1, but in two dimensions.

Unwrapped Ball 5D (UB) [36] is specific by the presence of a fraction and consists of 5 features which all influence the target value. Again, it does not fit the LCBF structure well.

A note on training and testing sampling. Originally (i.e. in the referenced articles), some of the benchmarks had different sampling for training and testing data than we present here. There are two modifications we have made:

- For Koza-1, originally there is no testing set, i.e. the same points are used both for training and testing. In order to make the results more descriptive, we decided to sample an independent testing set using the same procedure but producing more points (100).
- For S1, originally the training sampling is $E[0.05, 10, 0.1]$ and testing sampling is $E[-0.5, 10.5, 0.05]$. This means that the range of training data is smaller than the one of testing data. Because we want to focus on interpolation rather than extrapolation, we used the bigger of the two ranges, i.e. $[-0.5, 10.5]$ both for training and testing. We kept the original value of the grid spacing: 0.1 for training and 0.05 for testing.

4.3.2 Real-world benchmarks

The summary of the used real-world benchmarks is in Table 4.4. We used random 0.7/0.3 split for training/testing dataset.

Name	# of features	# of datapoints	Ref
ENC	8	768	[37–38]
ENH	8	768	[37–38]
CCS	8	1030	[39, 38]
ASN	5	1503	[38]

Table 4.4. Summary of the real-world benchmarks.

Energy Efficiency (ENC, ENH) [37] are datasets regarding energy efficiency of cooling (ENC) and heating (ENH) of buildings, acquired from the UCI repository [38]. They were already used as benchmarks in [15, 20], where the EFS and GSGP-Red methods respectively were introduced.

Concrete Compressive Strength (CCS) [39] is a dataset representing a highly non-linear function of concrete age and ingredients, acquired from the UCI repository [38]. This dataset was already used in [20] where GSGP-Red was introduced.

Airfoil Self-Noise (ASN), acquired from the UCI repository [38], is a dataset regarding the sound pressure levels of airfoils based on measurements from a wind tunnel. This dataset was also already used in [20].

4.3.3 Baseline algorithms

In order to provide reasonable baselines for the results of the four SR algorithms (and RdS), we have also run the same experiments with three classic machine learning algorithms. The implementations of all three ML algorithms were grabbed from the Python machine learning package `scikit-learn` [40–41].

Linear Regression (LR) is an ordinary least-squares multiple linear regression, without any form of regularization. The model is built just from the original input features.

Random Forest (RF) is an ensemble regression model made of a number of regression trees, each fitted to a slightly perturbed version of the training data.[1] Using the grid search, we tuned the following hyperparameters of the method:

- number of trees in the forest with possible values 5, 10, 50, 100, 200, and
- number of features to consider when looking for the best split with possible values N and $\sqrt{N}$, where N is the number of features of the dataset.

The grid search computes crossvalidation score for each grid point with 3-fold crossvalidation and selects the best settings[2]. For the purposes of computing time, we consider the grid search to be part of the training. In this view, one training run of RF has following steps:

1. for each grid point, run a 3-fold crossvalidation using the training data,
2. select the best grid point,
3. train the model parametrized by the best grid point using the whole training data.

Support Vector Regression (SVR)[3] with RBF kernel, combined with grid search in the following hyperparameters:

- C, the penalty parameter of the error term, with possible values 10^{-3}, 10^{-2}, 10^{-1}, 10^{0}, 10^{1}, 10^{2}, 10^{3}, and
- γ, the parameter of the RBF kernel, with possible values $0.01/N$, $0.1/N$, $1/N$, $10/N$, $100/N$, $1000/N$, where N is the number of features of the dataset.

The grid search works in the same way as in RF.

4.3.4 Settings and usage of the algorithms

The goal is to perform a comparison of the chosen methods as ready-to-use tools. Therefore, we didn't modify the code of the algorithms[4], and we left all of the settings at their default values.

Additionally, because the default function set of GPTIPS is very limited, we added a second version of GPTIPS, which we refer to as mGPTIPS, with the function set as close as possible to that of EFS without coding new functions, i.e. using only functions already available (either in MATLAB or in the GPTIPS package). This is possible because GPTIPS is easily configurable via a configuration file without the need to modify the code (in contrast to the other packages). Summary of the function sets of all compared methods is in Table 4.5.

Timeout. Both EFS and GPTIPS support a timeout after which the computation is terminated. We set it to 10 minutes for both methods. However, as will be seen in results in Table 4.11, except for RdS, all runs of all algorithms (including FFX and

[1] For details about the implementation and parameters see `http://scikit-learn.org/0.17/modules/generated/sklearn.ensemble.RandomForestRegressor.html`.

[2] For details about the implementation and parameters see `http://scikit-learn.org/0.17/modules/generated/sklearn.grid_search.GridSearchCV.html`.

[3] For details about the implementation and parameters see `http://scikit-learn.org/0.17/modules/generated/sklearn.svm.SVR.html`.

[4] The only exception is EFS: we changed the `round` variable to `false` (which was originally hard-coded to `true`) according to the issue we opened on the algorithm's GitHub repository, see `https://github.com/flexgp/efs/issues/1`.

function	GPTIPS	mGPTIPS, RdS	EFS	FFX	GSGP-Red
add	✓	✓	✓	✓[a]	✓
add3	✓				
sub	✓	✓	✓	✓[a]	✓
mult	✓	✓	✓	✓	✓
mult3	✓				
div		✓[p]	✓[p]	✓[b]	
AQ					✓
sqrt		✓[p]	✓[p]	✓[c]	
square		✓	✓		
cube		✓	✓		
quart			✓		
log		✓[p]	✓[p]	✓	
sin		✓	✓		
cos		✓	✓		
abs				✓	
max-hinge				✓	
min-hinge				✓	

[a] Only via top-level linear combination.
[b] Only via rational functions trick and sign of exponent of feature variable.
[c] Only of feature variable.
[p] Protected version.

Table 4.5. Function sets of individual algorithms. Functions **add3** and **mult3** are ternary addition and ternary multiplication, respectively. **AQ** is the Analytic Quotient operator [42] defined as $AQ(x_1, x_2) = \frac{x_1}{\sqrt{1+x_2^2}}$. Functions **max-hinge** and **min-hinge** are FFX's hinge functions defined as $max(0, x - thr)$ and $min(0, x - thr)$ respectively, where thr is a parameter.

GSGP-Red which have no support for timeout[5]) finished well within this amount of time. RdS has run for the whole 10 minutes of time.

Parameter values. We do not apply the grid search to tune the parameters of the SR methods because of several reasons:

- Some of the algorithms have very limited set of exposed settings and parameters (especially EFS, which has no exposed settings, see below). Thus, it would not be fair, if some algorithms went through tuning while the others would not.
- Grid search runs the method multiple times (number of grid points times the number of crossvalidation folds). The running time is (usually) acceptable in case of ML methods while it is often prohibitive in case of SR methods.
- Due to the highly stochastic nature of the majority of SR methods, the crossvalidation results have high variance (especially for a small number of folds usually used in grid search), i.e., the crossvalidation score is less reliable for scoring the configurations of SR methods (compared to the ML methods).

[5] FFX has a built-in 50s timeout for performing the fit of the elastic net. If the elastic net fails to fit within this time a constant model is returned for that particular fit. Note, however, that FFX fits the elastic net multiple times for each of the multiple runs (see Section 3.1.5) and there is no support for timeout of this combined run and returning the results obtained so far.

All three points, in our view, prohibit an effective use of grid search for the SR methods. Therefore, we use fixed parameter values, mostly identical to the defaults provided by the respective packages.

GPTIPS and mGPTIPS use identical default values of parameters, except for the extended function set for mGPTIPS. The most interesting parameters are

- population size is 100,
- number of generations is 150,
- tournament size is 10,
- fraction of elites is 0.15,
- max. tree depth is 4,
- max. number of genes is 4, and
- the initialization procedure is Ramped Half'n'Half.

EFS, except for the timeout, has no user-definable settings. Its settings are hardcoded and the most important are

- the number of composed features $q = 3p$, and
- the number new features created each generation $\mu = p$ where p is the number of features of the problem.

For details on the parameters, see the original paper [15].

FFX has no user-definable settings. It is worth to note, however, that the possible exponents for a variable are -1, -0.5, 0.5 and 1; it is thus impossible for the algorithm to create, for example, a quartic term, because FFX allows at most bivariate bases, i.e. bases of the form $a \cdot b$, where a and b are either variables raised to one of the exponents mentioned above, and/or a function of a variable. The greatest possible exponent is therefore 2, or -2, when a bivariate base has the form $x \cdot x = x^2$, or $x^{-1} \cdot x^{-1} = x^{-2}$.

GSGP-Red is very configurable via configuration files which allow the user to control all important aspects of the algorithm. The default values for the most interesting parameters are

- population size is 1000,
- number of generations is 250,
- tournament size is 10,
- probabilities of crossover and mutation are both 0.5, and
- initialization procedure is Ramped Half'n'Half.

Finally, RdS has only two parameters (except for the function set): maximum number of genes and maximum depth, which we set to 10 and 18 respectively in order to provide more "freedom" than the other algorithms have, because RdS is likely to evaluate more models as the only computational burden is the evaluation, and it is going to run for the whole 10 minutes.

Model training and selection. From each run of each algorithm, we need to get a single model as a result of said algorithm. EFS returns just a single model as a result, that best fits the training data. RdS behaves in the same way, as there is no set of models in the algorithm to begin with. We decided to use the same strategy also for FFX, GPTIPS and GSGP-Red. In case of FFX, which produces as its output a set of nondominated models with respect to performance and the number of bases, we selected the best model with respect to training MSE, which means the most complex model was selected. GPTIPS and GSGP-Red also return a population of models, from which the best one is chosen.

Choosing the model with minimal training set error might not be considered a good practice because of possible overfitting to the training set. Yet, we decided to do so, because of the following reasons:

- In four of the five methods, overfitting is constrained by setting hard limits on the expressional complexity (GPTIPS, EFS, FFX, RdS) and/or by putting soft emphasis on simpler models, like regularized linear regression (EFS, FFX), or parsimony pressure (GPTIPS)). GSGP(-Red) is reported to be resistant to overfitting because of the small steps it takes towards the optimum.
- Using the best solution from the population as the result is standard practice in GP, or EAs in general.

4.3.5 Model complexity constraints

Each of the selected algorithms handles the issue of resulting model complexity in a different way. GPTIPS has (user-defined) limits on the maximum number of nodes and/or maximum depth, and on the maximum number of bases. By default, there is a depth limit of 4, and maximum number of bases (not counting the intercept) is also 4. EFS computes the maximum number of bases from the number of input features; maximum number of nodes in a base is hard-coded to 5. The FFX procedure results in a maximum model depth of 5. In GSGP-Red, even though the tree reduction procedure is employed (which considerably reduces the size of the solutions), in principle, there is still no limit on the size just as in the original GSGP. In RdS, the individuals are limited in depth which we set to 18.

4.3.6 Testing environment

As we stated earlier in this chapter, we use existing open-source implementations of the algorithms except for RdS which we implemented ourselves. Table 4.6 summarizes where the implementations were retrieved from.

Algorithm	Implementation
GPTIPS	version 2, [43]
FFX	version 1.3.4, [44]
EFS	[32]
GSGP-Red	[45]
RdS	ours

Table 4.6. Sources of implementations of the algorithms.

All computations were performed on the same PC with Intel Core 2 Duo E6550 at 2.33 GHz, running 64-bit Ubuntu 15.04. The environments for the three algorithms were

- MATLAB version R2014a (8.3.0.532) 64-bit for GPTIPS,
- Java version 1.8.0_60-b27 for EFS and GSGP-Red,
- Python version 2.7.9 (built with GCC 4.9.2) for FFX, and
- Python version 3.4.3 (built with GCC 4.9.2) for RdS and the baseline algorithms.

4.3.7 Testing methodology

Each artificial dataset with uniform random sampling (i.e. the U-type sampling) was independently sampled 100 times. Artificial datasets with deterministic sampling (i.e. the E-type sampling) each have only a single instance which is used in all runs. Each real-world dataset was randomly and independently split 100 times into training and testing sets using 70 % and 30 % of the datapoints respectively.

Each algorithm was run once on each of the randomly sampled dataset instances (and 100 times on the deterministically sampled ones) producing a single model in each run. The accuracy and complexity of the resulting models are then aggregated and statistically compared. The only exception is the FFX algorithm on S1 and S2 datasets: these datasets are sampled deterministically (so there is only one instance for both these datasets) and the FFX algorithm is also deterministic, hence a single run is sufficient for these cases.

Evaluation. The performance metric is the RMSE on testing dataset. The model complexity metric is the number of nodes in the model. We define the number of nodes as the sum of the numbers of nodes across all basis functions of the model. We count only the expression trees themselves, i.e. we do not count the additional coefficients and operators related to the top-level linear combination produced by the linear regression approach used in the tested algorithms. These coefficients and operators are not counted because they are fully dependent on the bases themselves (their number) and counting them brings no interesting information[6]. FFX's *hinge functions*, having a form of $\max(0, x - thr)$ and $\min(0, x - thr)$, count as 5 nodes.

Differences between individual methods in terms of the testing RMSE and the model complexity are statistically evaluated using one-sided Mann-Whitney U-test (MWUT) [46–47] for each pair of algorithms with the Bonferroni correction with the significance level $\alpha = 0.01$.

4.4 Results and discussion

In this section we present the results. We discuss the global trends we recognize in the results as well as the notable patterns specific for each dataset. We also discuss the time demands of the methods, and the differences among SR and ML models.

4.4.1 Results

The results are presented both in the form of tables and plots. The tables show aggregated results for all algorithms and datasets, as well as the statistical significance. For an overall view, Table 4.7 shows median RMSE values on testing data for each algorithm and dataset. A detailed view on the performance, the statistical significance and the statistical comparison of all the algorithms is provided in the Table 4.8. A view on the complexity (number of nodes) of the produced models can be seen in Tables 4.9 and 4.10.

The plots show the performance and complexity of the individual algorithm runs, as well as the aggregated performance of each algorithm on both training and testing data in the form of box plots. The plots, one per dataset, can be seen in Figures 4.1 through 4.9.

[6] The number of nodes is used as a simple common measure of complexity across all the algorithms only for the purposes of the evaluation done here. The individual algorithms use their own measures of complexity in their search processes.

	SR methods						ML methods		
	GPTIPS	mGPTIPS	EFS	FFX	GSGP-Red	RdS	LR	RF	SVR
Koza-1	**0.000**	**0.000**	0.128	0.063	0.010	0.024	0.614	0.208	0.104
Korns-11	7.811	**7.749**	7.792	7.796	7.811	7.794	7.798	7.905	7.797
S1	0.291	0.111	0.269	0.294	0.221	0.230	0.302	**0.015**	0.060
S2	0.994	1.154	1.107	1.007	0.881	1.075	1.007	**0.228**	0.738
UB	0.141	0.114	0.076	0.083	0.118	0.159	0.188	0.069	**0.057**
ENC	2.907	2.278	1.640	1.791	3.172	3.011	3.252	1.633	**1.278**
ENH	2.538	1.717	0.546	1.046	2.726	2.691	2.926	**0.510**	0.674
CCS	8.762	7.178	6.429	5.986	8.798	8.314	10.52	**5.169**	10.03
ASN	4.138	4.003	3.623	3.580	12.14	4.471	4.816	**1.839**	6.054

Table 4.7. Median RMSEs on testing data. The best (smallest) values among SR and ML methods in each row are underlined. The best (smallest) value among all methods is typeset in bold.

	GPTIPS		mGPTIPS		EFS		FFX		GSGP-Red	
	rank	ssbt	rank	ssbt	rank	ssbt	rank	ssbt	rank	ssbt
Koza-1	1-2	EFSdLRV	1-2	EFSdLRV	7-8	L	4-5	LRV	3	EFLRV
Korns-11	2-8	R	1	GEFSdLRV	2-8	R	2-8	R	2-8	R
S1	7	FL	3	GEFSdL	5-6	GFL	8	L	4	GEFL
S2	4-5	EFd	7-9		7-9		6	Ed	3	GEFdL
UB	7	dL	5-6	GdL	3	GmFSdL	4	GmSdL	5-6	GdL
ENC	6-7	SL	5	GSdL	2-3	GmFSdL	4	GmSdL	8-9	
ENH	6	SL	5	GSdL	1-2	GmFSdLV	4	GmSdL	7-8	L
CCS	5-7	LV	4	GSdLV	3	GmSdLV	2	GmESdLV	5-7	LV
ASN	5	SdLV	4	GSdLV	2-3	GmSdLV	2-3	GmSdLV	9	

	RdS		LR		RF		SVR	
	rank	ssbt	rank	ssbt	rank	ssbt	rank	ssbt
Koza-1	4-5	LRV	9		7-8	L	6	LR
Korns-11	2-8	R	2-8	R	8		2-8	R
S1	5-6	GFL	9		1	GmEFSdLV	2	GmEFSdL
S2	7-9		4-5	EFd	1	GmEFSdLV	2	GmEFSdL
UB	8	L	9		2	GmEFSdL	1	GmEFSdLR
ENC	6-7	SL	8-9		2-3	GmFSdL	1	GmEFSdLR
ENH	7-8	L	9		1-2	GmFSdLV	3	GmFSdL
CCS	5-7	LV	8-9		1	GmEFSdLV	8-9	L
ASN	6	SLV	7	SV	1	GmEFSdLV	8	S

Table 4.8. Statistical ranking of RMSEs. Left columns show the rank of the algorithm. The title of right columns, "ssbt", stands for *statistically significantly better than*, and they show algorithms that were statistically significantly worse as judged by the MWUT. The significance level is $\alpha = 0.01$ which is reduced by a Bonferroni correction for 36 tested pairs resulting in $\alpha \approxeq 0.00028$ for each tested pair. The individual algorithms are denoted in the following way: **G** for GPTIPS, **m** for mGPTIPS, **E** for EFS, **F** for FFX, **S** for GSGP-Red, **d** for RdS, **L** for LR, **R** for RF, and **V** for SVR.

	GPTIPS	mGPTIPS	EFS	FFX	GSGP-Red	RdS
Koza-1	33	14	**11**	35	7703	20
Korns-11	63	17	69	14	3221	**11**
S1	52	23	12	**10**	21342	11
S2	53	25	28	**1**	10034	8
UB	36.5	**10.5**	66	105	7559	14
ENC	48	2	108	136	6883	**20**
ENH	47.5	26	105	146	7047	**21**
CCS	43	23	108	474.5	6009	**17**
ASN	58	30	67	52.5	33320	**21.5**

Table 4.9. Median number of nodes for each algorithm and dataset. The best value in each row is highlighted.

	GPTIPS		mGPTIPS		EFS		FFX		GSGP-Red		RdS	
	rank	ssbt	rank	ssbt	rank	ssbt	rank	ssbt	rank	ssbt	rank	ssbt
Koza-1	4-5	S	2	GFS	1	GmFSd	4-5	S	6		3	FS
Korns-11	4-5	S	2-3	GES	4-5	S	2-3	GES	6		1	GmES
S1	5	S	4	GS	2-3	GmS	1	GmES	6		2-3	GmS
S2	5	S	3-4	GS	3-4	GS	1	GmESd	6		2	GmES
UB	3	EFS	1-2	GEFS	4	FS	5	S	6		1-2	GEFS
ENC	3	EFS	1-2	GEFS	4	FS	5	S	6		1-2	GEFS
ENH	3	EFS	1-2	GEFS	4	FS	5	S	6		1-2	GEFS
CCS	3	EFS	2	GEFS	4	FS	5	S	6		1	GmEFS
ASN	3-4	ES	2	GEFS	5	S	3-4	ES	6		1	GmEFS

Table 4.10. Statistical ranking of complexities (number of nodes). Left columns show the rank of the algorithm. The title of right columns, "ssbt", stands for *statistically significantly better than*, and they show algorithms that were statistically significantly worse as judged by the MWUT. The significance level is $\alpha = 0.01$ which is reduced by a Bonferroni correction for 15 tested pairs resulting in $\alpha \approxeq 0.00067$ for each tested pair. The individual algorithms are denoted in the following way: **G** for GPTIPS, **m** for mGPTIPS, **E** for EFS, **F** for FFX, **S** for GSGP-Red, and **d** for RdS.

4.4.2 Discussion on error and complexity

In this subsection we discuss the results from the point of view of the achieved RMSE and model complexity in terms of the number of nodes.

Koza-1. As can be seen from Table 4.7 and Figure 4.1, GPTIPS and mGPTIPS were the only methods that were able to achieve zero error. GPTIPS with the default function set found such model in all runs, although needing more nodes for that. Enriching the function set (mGPTIPS) enables the method to find simpler models also with optimal performance, but – due to a larger search space – it sometimes fails to find the optimum. RdS was able to find a zero-error model, but did so in only a few runs and otherwise had inconsistent performance which is expected since the algorithm is just a random search.

FFX and EFS are worse, both reaching RMSE of the order of 10^{-2} with no significant difference between them (see Table 4.8), partially due to the large range of RMSE values produced by EFS. The non-zero error is caused by the regularization used in these

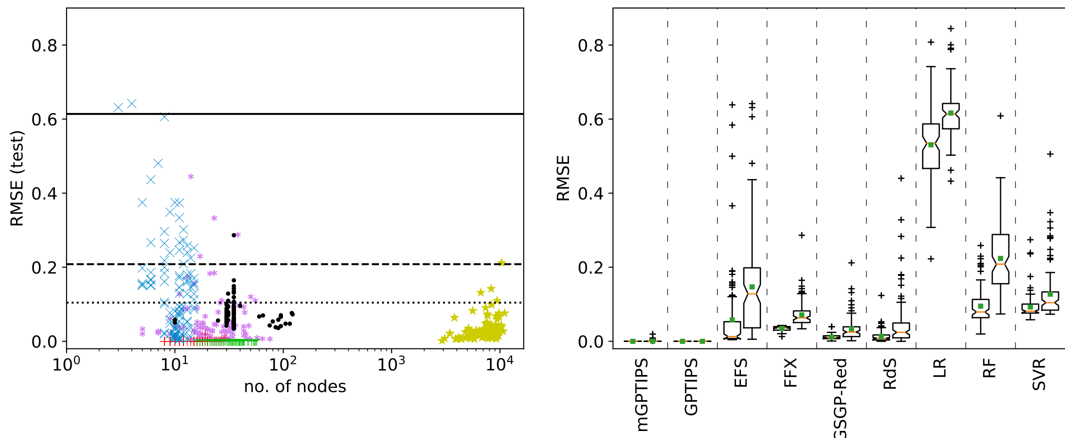


Figure 4.1. Complexity-performance plots (left) and box plots of training and testing errors (right) for the Koza-1 dataset. Legend: individual runs of Y GPTIPS, + mGPTIPS, × EFS, • FFX, ⋆ GSGP-Red, ∗ RdS, median RMSE of — LR, - - - RF, · · · SVR.

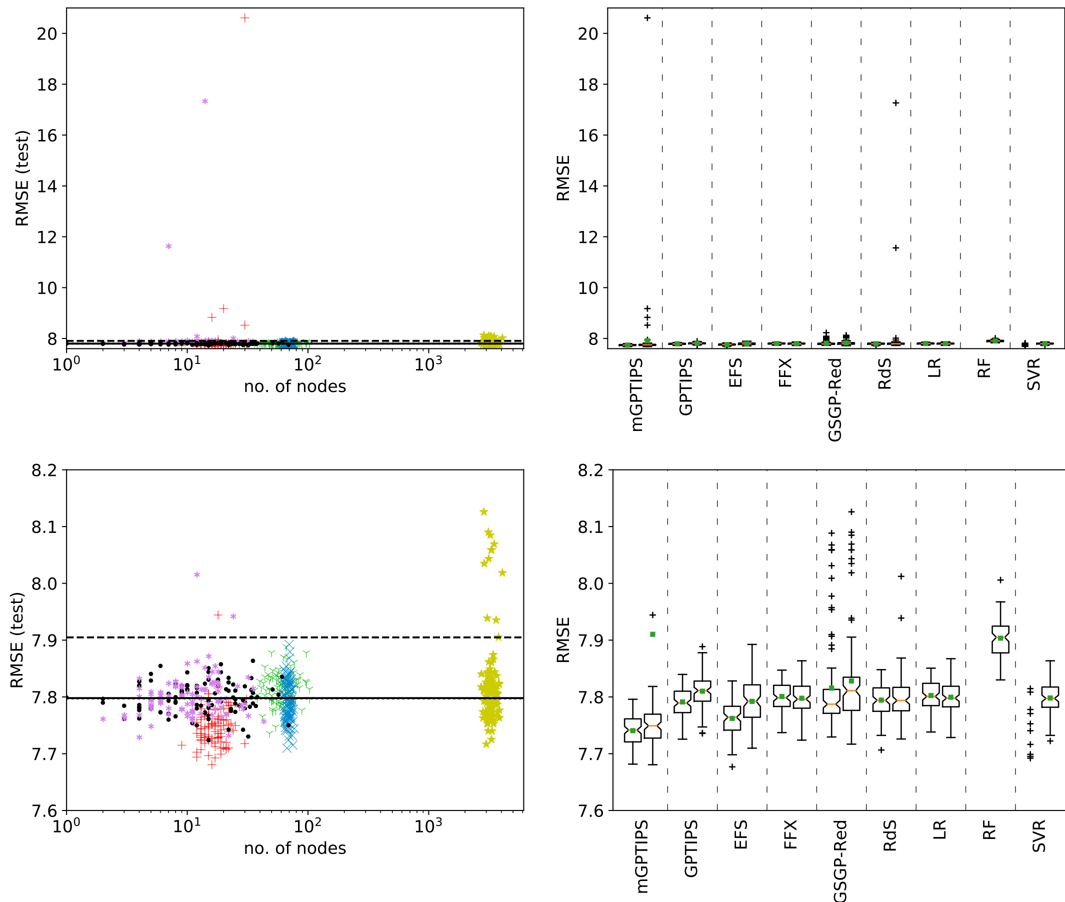


Figure 4.2. Complexity-performance plots (both left) and box plots of training and testing errors (both right) for the Korns-11 dataset. The upper plots display the whole results, the lower ones zoom on the dense area around RMSE = 7.8. Legend: individual runs of Y GPTIPS, + mGPTIPS, × EFS, • FFX, ⋆ GSGP-Red, ∗ RdS, median RMSE of — LR, - - - RF, · · · SVR.

methods – EFS indeed found the optimal bases, but their coefficients are not equal to 1.

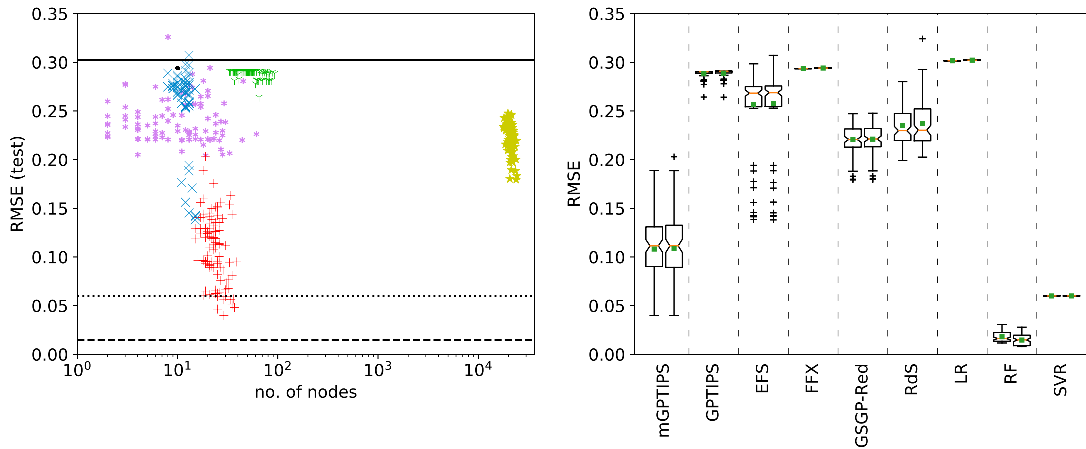


Figure 4.3. Complexity-performance plots (left) and box plots of training and testing errors (right) for the S1 dataset. FFX has only a single point because both the sampling of this dataset and FFX are deterministic. Legend: individual runs of Y GPTIPS, + mGPTIPS, × EFS, • FFX, ⋆ GSGP-Red, ∗ RdS, median RMSE of — LR, - - - RF, · · · SVR.

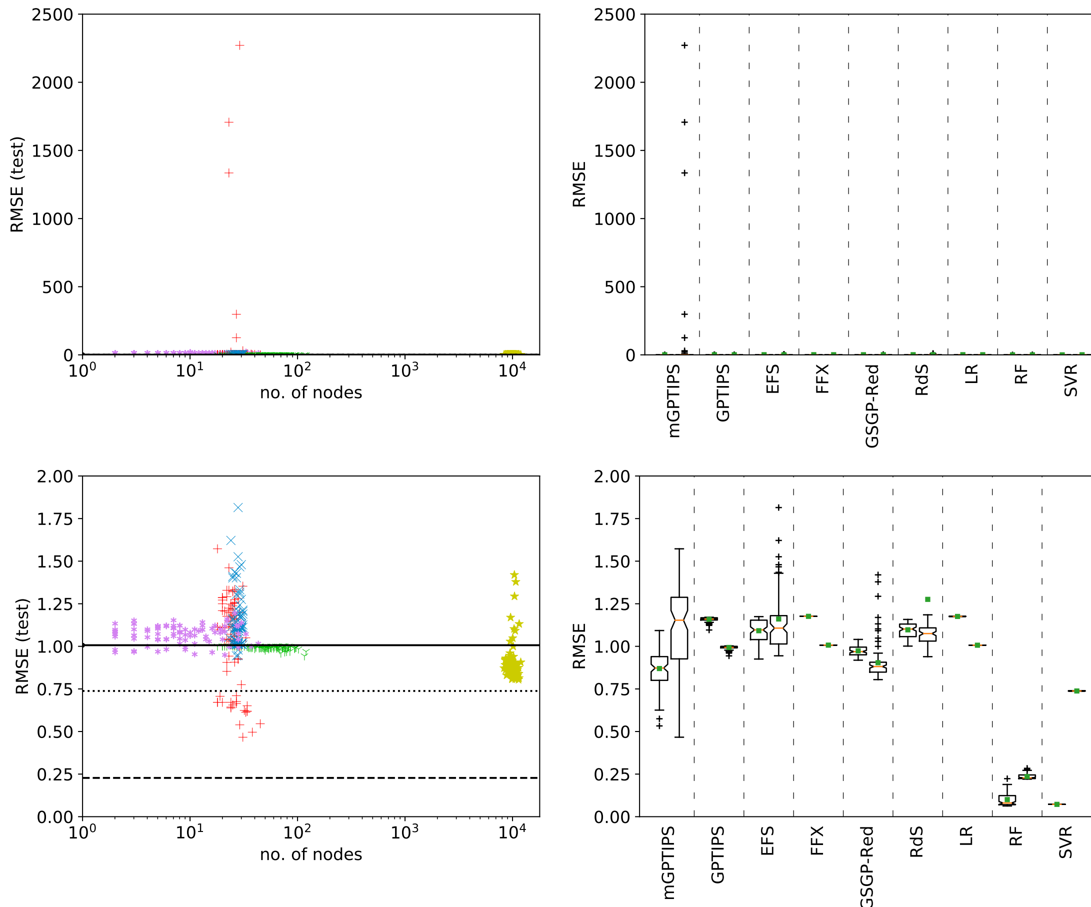


Figure 4.4. Complexity-performance plots (both left) and box plots of training and testing errors (both right) for the S2 dataset. FFX has only a single point because both the sampling of this dataset and FFX are deterministic. The upper plots display the whole results, the lower ones zoom on the dense area around RMSE = 1. Legend: individual runs of Y GPTIPS, + mGPTIPS, × EFS, • FFX, ⋆ GSGP-Red, ∗ RdS, median RMSE of — LR, - - - RF, · · · SVR.

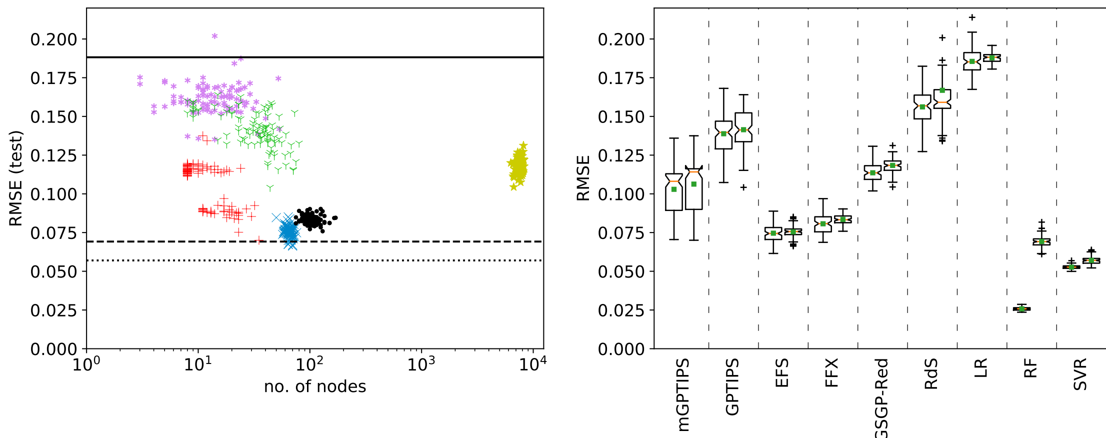


Figure 4.5. Complexity-performance plots (left) and box plots of training and testing errors (right) for the UB dataset. Legend: individual runs of Y GPTIPS, + mGPTIPS, × EFS, • FFX, ⋆ GSGP-Red, ∗ RdS, median RMSE of — LR, - - - RF, · · · SVR.

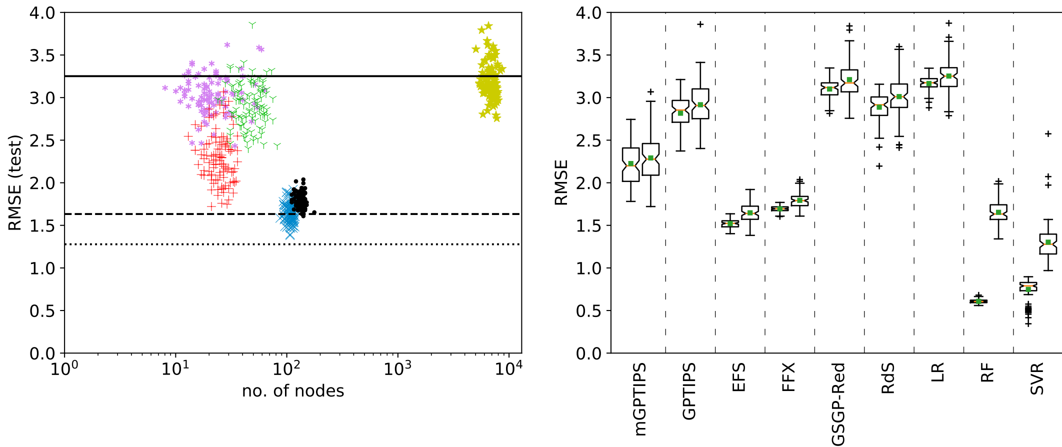


Figure 4.6. Complexity-performance plots (left) and box plots of training and testing errors (right) for the ENC dataset. Legend: individual runs of Y GPTIPS, + mGPTIPS, × EFS, • FFX, ⋆ GSGP-Red, ∗ RdS, median RMSE of — LR, - - - RF, · · · SVR.

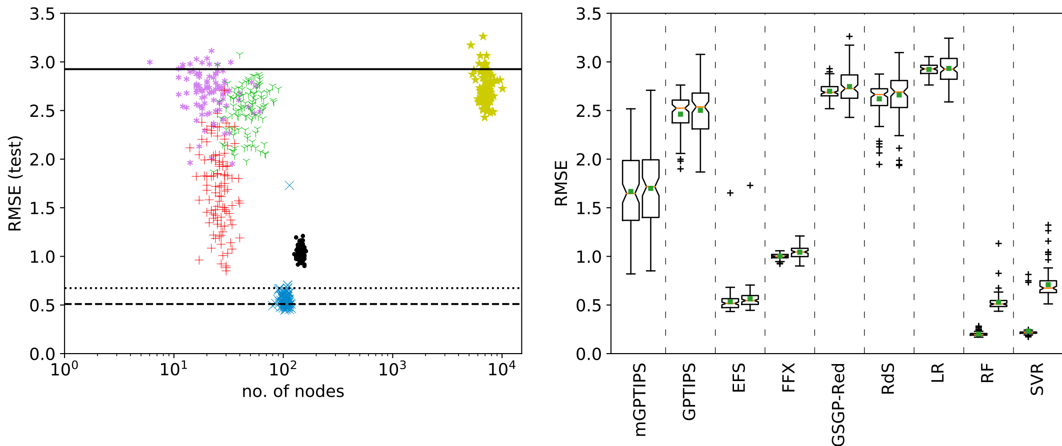


Figure 4.7. Complexity-performance plots (left) and box plots of training and testing errors (right) for the ENH dataset. Legend: individual runs of Y GPTIPS, + mGPTIPS, × EFS, • FFX, ⋆ GSGP-Red, ∗ RdS, median RMSE of — LR, - - - RF, · · · SVR.

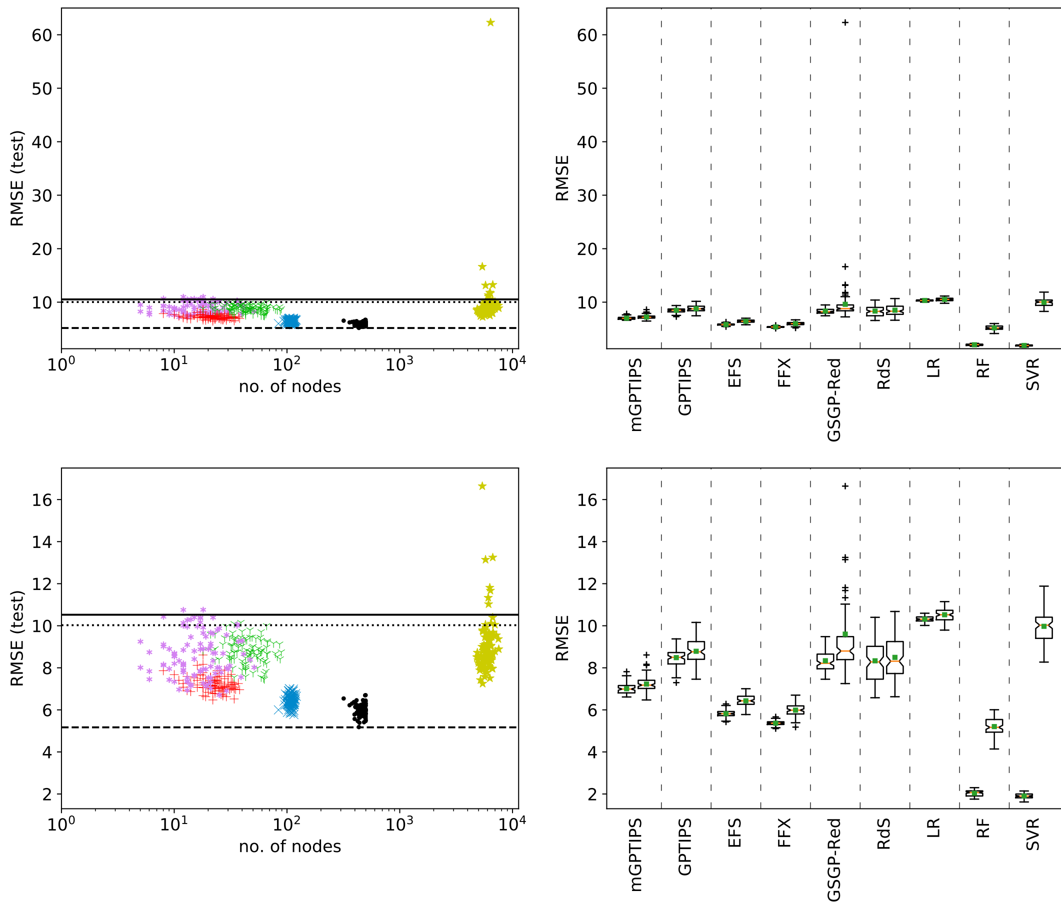


Figure 4.8. Complexity-performance plots (left) and box plots of training and testing errors (right) for the CCS dataset. The upper plots display the whole results, the lower ones zoom on the dense area around RMSE = 8. Legend: individual runs of Y GPTIPS, + mGPTIPS, × EFS, • FFX, ⋆ GSGP-Red, ∗ RdS, median RMSE of — LR, - - - RF, ··· SVR.

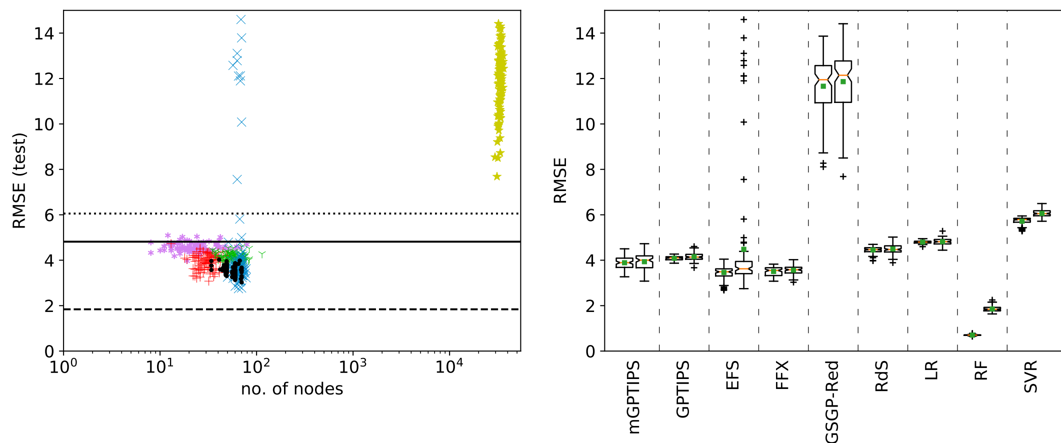


Figure 4.9. Complexity-performance plots (left) and box plots of training and testing errors (right) for the ASN dataset. Legend: individual runs of Y GPTIPS, + mGPTIPS, × EFS, • FFX, ⋆ GSGP-Red, ∗ RdS, median RMSE of — LR, - - - RF, ··· SVR.

GSGP-Red is slightly more accurate than FFX both for training and testing data but it explodes in the complexity of the models, which is a global trend that can be

seen over all datasets. The high complexity is expected of a GSGP-based algorithm, even though the –Red extension simplifies the models significantly.

FFX and GPTIPS tend to construct significantly more complex models (see Tables 4.9 and 4.10) than EFS and mGPTIPS – this is most likely caused by the fact that they are unable to effectively create the 4th power, and therefore need to compensate for it by creating a lot of bases.

For Koza-1, the SR models are better than or comparable to the tuned ML models.

Korns-11. This dataset comes from a quickly oscillating function (see Table 4.2) with a constant range of values. The datapoints look very much like samples from a constant function with noise. As can be seen from Tables 4.7, 4.8, and Figure 4.2, all the methods (SR and ML) provide models of comparable performance. The best for this problem is mGPTIPS which is better than the others from the statistical point of view despite the outliers; the real importance of the difference is, however, questionable. The fact that some algorithms (mGPTIPS and EFS) had sin and cos functions in their function set does not seem to affect the result. We argue that that is caused by the already mentioned noisy look of the data – the sin/cos cannot match right away, they would have to have the right argument from the start which is very unlikely and even a small difference in the argument can lead to dramatic errors w.r.t. the training data.

FFX and mGPTIPS produced models with significantly smaller number of nodes than GPTIPS and EFS (see Tables 4.9 and 4.10). Even though FFX is deterministic, the complexity of its models varies highly. The only possible cause are the differences in the individual dataset samplings themselves. Somewhat unexpected is the fact that it influences FFX so much compared to the stochastic EFS. Note, however, that despite the larger variance in complexities, the overall complexity of FFX models is still significantly lower than that of EFS models. GSGP-Red produces several orders of magnitude more complex models than the other methods. RdS has performance very similar to that of FFX in both metrics with the exception of a couple of outliers in the performance metric.

S1. As can be seen from Table 4.7 and Figure 4.3, GPTIPS, which has the most limited function set among the compared methods, produces complex models with relatively large errors. FFX produced a simpler model (10 nodes only) with comparable error. The complexity of EFS models is comparable to FFX, but EFS tends to produce more accurate models. The best trade-off is provided by mGPTIPS models which are significantly more accurate, with complexities only slightly worse than those of EFS. The performance of GSGP-Red is “in between” the performances of mGPTIPS and EFS but with significantly more complex models. Note that FFX was run only once since it is a deterministic algorithm and there is only a single instance of this dataset. RdS is performs better than GPTIPS in both complexity and performance, and than EFS in performance, with complexities being spread over a wider range.

An interesting point is that among the SR algorithms, the two best were mGPTIPS and EFS, both having sin and cos functions available (and they are used in the models), compared to GPTIPS and FFX which don’t have these functions. The S1 dataset is defined by a function containing both sin and cos functions.

The performance of SR models on this benchmark is better than pure linear regression, but worse than RF and SVR.

S2. For this problem, the only algorithm that produced models discernibly better than a constant function from a practical point of view was RF. Out of the SR methods, only FFX was able to provide the constant model with only a single node, as can be seen

in Table 4.9 and Figure 4.4. Default GPTIPS provides models with comparable performance (yet statistically better than FFX), but with much larger complexity. Some models of mGPTIPS are in fact able to reach better performance, but sometimes also much worse (by several orders of magnitude). EFS provides results similar to mGPTIPS, but more consistent. GSGP-Red is comparable to other evolutionary methods but with orders of magnitude more complex models. RdS behaved similarly to other methods but generally with lower complexities and smaller variance in performance than that of mGPTIPS and EFS.

UB. Except for LR and RdS, the default GPTIPS is the least accurate solver here, as can be seen in Table 4.7 and Figure 4.5, and also statistically confirmed in Table 4.8. Enlarging the function set allows mGPTIPS to find not only more accurate but also simpler models, but still not as good as those provided by EFS and FFX. GSGP-Red's performance is in between that of GPTIPS and mGPTIPS, with very high model complexity. The most accurate SR algorithms for this problem are EFS and FFX, with EFS generating models with lower number of nodes than FFX. Both EFS and FFX, however, produce more complex models than (m)GPTIPS and RdS.

Similarly to S1, SR methods are better than pure LR, but worse than SVR and RF.

ENC, ENH. As can be seen in Figures 4.6 and 4.7, the pattern of the results is similar for both datasets in terms of both the accuracy and complexity of the models, which can also be seen in Tables 4.7–4.10. The results of GPTIPS are dominated both in accuracy and complexity by mGPTIPS, the results of FFX are dominated by EFS in a similar fashion. GSGP-Red is dominated by all other methods except for LR. RdS exhibits comparable performance as that of GPTIPS but with lower complexities. EFS and mGPTIPS provide a good compromise with EFS producing more accurate models, while mGPTIPS producing simpler models.

RF and SVR are comparable or better than the best of SR methods, except for EFS, in terms of accuracy.

CCS. In this dataset, a similar pattern among SR algorithms as in ENC and ENH is also present, except that the accuracies of EFS and FFX are flipped and the differences are not as strong, as can be seen in Figure 4.8 and Tables 4.7 and 4.8.

From the complexity point of view, however, the ENC/ENH pattern remains: mGPTIPS and RdS provide the simplest models, followed closely by GPTIPS. EFS produces just over a hundred nodes, FFX explodes with four to five hundred of nodes and GSGP-Red is in the area between one and ten thousand nodes. The high number of nodes in FFX's models is caused by the majority of bases being the *hinge functions* which carry high complexity.

RF models are only slightly, but significantly better than those of the best SR algorithms – FFX and EFS. All SR algorithms produce on average better models than pure linear regression. Note, however, the failure of SVR on this dataset – it is better than LR by only a small margin. Having the best training errors and much worse testing errors, SVR is suspect from overfitting here.

ASN. Figure 4.9 shows that all the SR methods except for GSGP-Red perform similarly in terms of RMSE but by statistical comparison they are different (see Table 4.8). EFS produced a number of outliers (some actually worse than a pure linear model) and is thus less reliable. GSGP-Red is clearly the worst algorithm here, not producing a single model better than LR.

The simplest models are produced by RdS, followed by mGPTIPS and then by FFX and GPTIPS which are statistically indifferent (Table 4.10), and then closely by EFS. As in all other datasets, GSGP-Red produces the largest models.

RF again produced the most accurate models. SVR failed again, with both the training and testing errors larger than the errors of LR. The explanation may lie in the dataset which may be unsuitable for SVR modelling. Another reason may be the fact that SVR optimizes its ε-intensive loss function, and not RMSE.

4.4.3 Global trends

Across all datasets we can see that none of the compared SR algorithms was the best everywhere, both from the performance and complexity points of view. We can see that EFS and FFX perform quite well on real-world datasets and the UB dataset, but not as well on the other artificial datasets. This suggests that for certain class of real-world problems the inability to work with internal constants, of which only (m)GPTIPS and indirectly GSGP-Red are capable, is not crucial and can be compensated by a linear combination of sufficiently large number of features.

Across all datasets, EFS and FFX methods are very consistent, meaning that the clusters in complexity-performance space are compact and without too many outliers. This fact might be important in applications where consistency of the produced models is an issue. In contrast to (m)GPTIPS, this may be the results of the regularized learning employed in EFS and FFX.

(m)GPTIPS tends to have a higher spread of either complexity or accuracy or both (except on Korns-11 where all the algorithms are similarly inconsistent). We argue that this is caused by the approach being close to vanilla GP based on population of models, in contrast to the population of features of EFS and deterministic generation of features in FFX.

The performance of RdS, when compared to the other SR methods, is interesting. Note that the RdS algorithm is supposed to be a just a baseline – it randomly generates LCBF models and applies LR on the bases but nothing more. It does not use previously found models to create future ones, it only remembers the best generated model encountered. And despite this simplicity, the performance is always in the vicinity of the non-random methods, sometimes even better than (some of) them. This suggests, that the only non-random element in the RdS algorithm – the linear regression – is quite powerful in itself when provided with suitable features.

The comparison of SR methods with conventional ML approaches (with tuned hyperparameters) shows that the SR methods are not the best performers. In the majority of cases, the SR methods were better than pure LR models, but were worse than RF or SVR models. For many datasets it can also be observed that the differences between training and testing errors were much larger for RF and SVR models than for SR models. We thus hypothesize that with the default settings of the SR methods may be too constraining and possibly produce underfitted models, while the settings found by the grid search for RF and SVR may result in somewhat overfitted models.

Another phenomenon worth noting is that many times the SR methods were worse than pure LR despite all of them use LR too. However, in (m)GPTIPS the LR is performed on the evolved bases only, lacking the original feature variables. In EFS and FFX the feature variables are always present, but the LR is regularized.

4.4.4 Comparison with previous research

The non-dominance of the SR methods seems to be in contradiction to the original papers of EFS [15] and FFX [16] and research done with GPTIPS [12] which usually found the SR models to be better than conventional ML ones. However, in each of these papers the methods were compared in a different way with different settings of both the SR methods and the traditional ML algorithms. The differences are discussed below.

The original paper on EFS [15] shows that EFS is overall better than most of the ML methods it was compared with, while in this work we found that one of the conventional ML methods is better on all problems except Koza-1 and Korns-11, which might be seen as a contradiction. The settings of EFS in this paper are the same as those used in [15]. However, the original paper compared EFS with different ML algorithms besides LR (for which our results are consistent) and also on different datasets except ENC and ENH. In [15], EFS was beaten by a feed-forward neural network (FFNN) on two datasets and was very close to FFNN on the others. We thus hypothesize that the different conclusion in this case is caused by using a different set of competing ML methods. It could well be that our settings of RF and SVR allowed for better regression models than their settings for FFNN, but the authors gave only a little details on the settings of FFNN (they stated that they used 10 neurons in a hidden layer and the toolbox used, but provide no information about activation functions, learning algorithm, etc.).

In [16], FFX was shown to outperform many ML methods, including LR and SVR, on six 13-dimensional problems, which, again, seems to contradict our results. However, it is important to note three facts:

1. all six regression problems in the original paper were some performance characteristics of a CMOS operational transconductance amplifier, i.e., they come from a very specific domain;
2. the paper does not describe the settings of the ML methods (including the SVR) in any detail;
3. the settings of FFX in those experiments were different than those provided as the defaults, namely the set of possible exponents included also -2 and 2 (see Section 4.3.4 for setting used in this work).

Due to these reasons, the different conclusions do not have to contradict each other.

Some results comparing GPTIPS to conventional ML methods can be found in [12]. GPTIPS was compared with SVR and artificial neural network (ANN) and GPTIPS was reported to be better than both. In that research, SVR used a linear kernel and its parameters were determined by a combination of simulated annealing and grid search. The experiments were done on only one 2-dimensional dataset from the domain of modelling soil-water retention curves. GPTIPS used in that research was "version 1" [48], while we used "version 2" [33], and the used algorithm settings were different from the default ones (larger population, smaller tournament, larger limits on tree depth and number of genes). It is thus well possible that the used parameter settings were chosen with respect to the analysed dataset, but we cannot rule out the possibility that GPTIPS with less limiting settings than the default ones would provide better results even in our study.

In general, our results differ from the previous research, but this seems to be caused by testing on different datasets, comparing to different ML algorithms, and using different settings of both the SR and the ML algorithms.

4.4.5 Discussion on running time

Evaluation of the running time of the algorithms is presented to provide at least some view into this aspect of the algorithms. The running times are presented in Table 4.11. The results are also influenced by the implementation language and running environment (FFX runs in Python, EFS and GSGP-Red in Java, GPTIPS in MATLAB). Because of this, the running times are only informative and do not necessarily represent the real computational complexity of the algorithms.

	GPTIPS	mGPTIPS	EFS	FFX	GSGP-Red	LR	RF (10)	SVR (42)
Koza-1	44.98	33.51	**0.33**	1.71	27.66	<0.01	2.85	0.34
Korns-11	101.91	90.18	16.38	**7.86**	319.04	<0.01	71.87	1138.74
S1	58.87	44.37	**0.38**	6.85	128.58	<0.01	2.88	0.54
S2	58.11	48.82	1.26	**0.56**	56.53	<0.01	2.97	5.71
UB	48.05	36.27	6.38	**6.15**	61.61	<0.01	6.05	6.67
ENC	57.40	51.09	**24.96**	109.63	40.42	<0.01	3.02	7.59
ENH	59.63	53.23	**25.42**	129.83	41.04	<0.01	2.96	11.61
CCS	59.22	51.15	**19.29**	30.58	43.08	<0.01	4.73	5.32
ASN	68.00	57.86	**9.43**	22.20	269.70	<0.01	4.17	12.90

Table 4.11. Median running times of the algorithms per dataset (in seconds). The Random Search (RdS) algorithm is not displayed because it always ran for the whole 10 minutes of available time. For RF and SVR, the number in the parentheses denotes the number of points of the grid search which is included in the running time. The fastest running times among the SR algorithms are emphasized.

Based on the wall-clock time, from the SR algorithms, (m)GPTIPS and GSGP-Red tend to run for the longest time (tens of seconds), with the exception of ENC and ENH datasets, where FFX was even slower. Runtime of EFS follows the number of features in the dataset: with Koza-1 and S1 (1D) requiring the least time, S2 (2D) requiring some more time, followed by Korns-11, UB, and ASN (5D), and finally ENC, ENH, and CCS (8D) requiring the most time. Note that EFS determines the number of bases from the number of features.

The time demands of the SR methods usually depend only linearly on the number of training examples (since they are used typically only to compute the value of evaluation function). Conventional ML methods may have much worse dependency on the number of training examples. This difference is pronounced in our study in case of the SVR algorithm (which needs to compute the kernel matrix) and the Korns-11 benchmark which has a large training set, where SVR is by far the slowest algorithm. In other cases, the time required to find a symbolic model was about an order of magnitude greater than the time to tune and train a conventional ML model (with the exception of pure LR which is of course the fastest among the algorithms), but still within one minute of time in most cases.

4.5 Comparison with results from omitted algorithms

As we have stated at the start of this chapter, we selected only some of the state-of-the-art algorithms, because some of them were published only after the research presented here was performed. In this section we briefly review research performed with these

newer algorithms and, where possible, try to estimate how would they compare, if they were included in the comparison.

Interestingly, many of the algorithms we reviewed in Chapter 3 have not been tested on real-world problems by their respective authors. It is therefore next to impossible to estimate how would these algorithms compare to those in our study. The algorithms that were in some way tested on real-world problems are ParetoGP, AI Feynman and FEAT[7].

ParetoGP. In [23], the algorithm is tested on a "polymer reactivity problem" but the authors do not provide the source of the dataset; in [24], the authors use the "Tower" problem, a gas chromatography measurement of the composition of a distillation tower, but the authors also do not point to the source of the dataset. Even neglecting the unavailability of the datasets, the authors compared their algorithm only to standard GP. It is therefore impossible to estimate how would ParetoGP perform compared to other algorithms.

AI Feynman. The authors tested using formulas from physical laws which could be argued to be real world. However, they are still known formulas which were just used to generate data for the algorithm to solve. Our notion of a real-world data is that the underlying process is not known and the data comes from measurements of a real-world system.

FEAT. Testing of this algorithm has been the most compatible with our study. Both the original version [27] and the extended version with semantic-aware operators [29] were examined on 100 regression datasets from PMLB [49]. This has shown FEAT (or its versions with the semantic-aware operators) to be competitive with XGBoost and to outperform multilayer perceptron, random forests, GSGP and other methods.

The extended version also went through a hyperparameter tuning which was performed on a number of datasets we used in our comparison here, specifically the UB, ENC, ENH, CCS, and ASN datasets. Table 4.12 shows the performance of the best variants of FEAT on these datasets compared to the SR algorithm from our study which had the best median RMSE on that dataset. However, the authors did not present specific numbers but only plots of R^2 score and we have to read from them. Also, they used 5-fold crossvalidation while we used repeated training/testing split, therefore the translation from R^2 to RMSE (which is the metric we used) provides just a rough picture.

To translate their reported R^2 score, we basically invert the definition of R^2 and transform it to RMSE:

$$
\begin{aligned}
R^2 &= 1 - \frac{\sum_{i=1}^{n}(\hat{y}_i - y_i)^2}{\sum_{i=1}^{n}(\bar{y}_i - y_i)^2} \\
R^2 - 1 &= -\frac{\sum_{i=1}^{n}(\hat{y}_i - y_i)^2}{\sum_{i=1}^{n}(\bar{y} - y_i)^2} \\
(R^2 - 1)\sum_{i=1}^{n}(\bar{y} - y_i)^2 &= -\sum_{i=1}^{n}(\hat{y}_i - y_i)^2 \\
(1 - R^2)\sum_{i=1}^{n}(\bar{y} - y_i)^2 &= \sum_{i=1}^{n}(\hat{y}_i - y_i)^2
\end{aligned}
$$

[7] SSR (see Section 3.1.1) and MRGP (see Section 3.1.3) were also subject to testing on real-world data, but we deem them surpassed by MGGP and EFS respectively.

$$\begin{aligned}
\frac{1}{n}(1-R^2)\sum_{i=1}^{n}(\bar{y}-y_i)^2 &= \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i-y_i)^2 \\
\sqrt{\frac{1}{n}(1-R^2)\sum_{i=1}^{n}(\bar{y}-y_i)^2} &= \sqrt{\frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i-y_i)^2} \\
\sqrt{\frac{1}{n}(1-R^2)\sum_{i=1}^{n}(\bar{y}-y_i)^2} &= RMSE
\end{aligned} \tag{4.1}$$

where n is the number of data points, $\hat{y}_i$ is the model estimate for the i-th data point, y_i is the true target value of the i-th data point, and $\bar{y}$ is the mean true target value of all data points. The sum on the left-hand side of Equation (4.1) is a quantity which we can estimate. Since we do not know which specific data points were included in the crossvalidation folds, we compute the quantity over the whole dataset, which, we think, is good enough approximation for a rough comparison.

dataset	best FEAT median R^2	rough equiv. RMSE	best SR alg.	best SR alg. median RMSE
UB	0.9195	0.06	EFS	0.076
ENC	0.973	0.33	EFS	1.64
ENH	0.9976	0.1	EFS	0.55
CCS	0.858	53.47	FFX	5.97
ASN	0.726	3.57	FFX	3.58

Table 4.12. Rough comparison of FEAT from tuning experiments from [29]. The column “rough equiv. RMSE” roughly shows what the RMSE would have been if FEAT achieved the reported R^2 score in our testing environment.

From these results it seems that FEAT would pose very strong competition to the algorithms we have tested.

4.6 Answers to research questions

In Section 4.2, we have stated three research questions. Armed with the experimental results, we answer those questions here.

RQ1: What are the performance differences between the algorithms? The results show that none of the benchmarked algorithms is superior to the others with their default settings. GPTIPS shows more varying performance, possibly given by the close relation to vanilla GP, while EFS and FFX provide more consistent results.

RQ2: How do the algorithms compare to reasonable baselines? The results of the comparison with random search show that the SR algorithms are overall better than the random search, which is an expected behaviour. However, the differences are not as large as one might have expected which, we argue, is the effect of the versatility of the LCBF form and the linear regression which is able to create good models even when the underlying bases are by themselves not of great quality.

Regarding the comparison with conventional ML algorithms, the results show that the SR algorithms are no silver bullet – in the majority of cases, an ML algorithm was more successful than any of the SR methods, especially RF. Note, however, that the ML algorithms each went through a grid search for the optimal parameter settings while

the SR algorithms were used without any tuning (see Section 4.3.4 for reasons). Also, even in cases when the ML methods were statistically significant, the gap between them and the SR methods was not large. Moreover, the SR methods have the benefit of producing symbolic models which may be an important asset in circumstances when not only prediction accuracy is of interest but also the understanding of the inner structure of the model is required. Last but not least, RF and SVR have shown greater discrepancy between training and testing errors, hinting that they may be more susceptible to overfitting.

Based on the results we obtained we think that classic ML methods are often as good as the SR methods with default parameter settings, or better, and they provide the results faster and often are more reliable.

RQ3: What algorithm should we choose for further experimentation? Based on the results, we choose GPTIPS, or rather the underlying algorithm MGGP. Its performance was not the best but was still competitive with the other algorithms, while it is the least constrained algorithm.

The reasons why we have not chosen the other algorithms are following:

- FFX severely restricts the class of models it can produce and easing this restriction would result in combinatorial explosion, should it still be the same algorithm.
- EFS, while it has good performance, is built around the idea is that the features are built only constructively which allows it to not reevaluate already evaluated subtrees. The extensions we want to implement require reevaluation of subtrees, and doing so with EFS would require rewrite of major parts of the algorithm, possibly rendering it significantly less effective than it is.
- GSGP-Red had poor performance on the real-world datasets, and the models produced by the algorithm are about two orders of magnitude larger than those produced by the other methods.

4.7 Summary and conclusions

In this chapter we compared four recent methods for symbolic regression: MGGP, EFS, FFX, and GSGP-Red, plus a baseline in the form of RdS, a random search of a model in the form of LCBF with top-level linear parameters fitted with LR. All the algorithms produce models in the form of LCBFs. Two of those methods, EFS and FFX, use Pathwise Regularized Learning, while MGGP (and RdS) uses classic multiple linear regression to determine the linear coefficients of the resulting model. EFS and MGGP are stochastic methods based on GP operators of mutation and crossover, while FFX is a completely deterministic method.

We used publicly available implementations of the algorithms, with their default settings and without modifications of their implementations. Since the default settings of GPTIPS, an implementation of MGGP, contain a very limited function set, we added an extra configuration with a function set closer to the one used by EFS.

We compared the algorithms with three classic ML algorithms – multiple linear regression, random forests, and support vector regression. The results have shown that the classic ML methods are, overall, the best performers, and that there is no single SR algorithm that would outperform all the others. Despite not being the best performer, we have selected MGGP as the algorithm to continue our research with, as it is the most extensible one, while its performance was still competitive.

Chapter 5
Feature space transformations for genetic programming

In some SR tasks, the underlying function could be modelled more easily, if the algorithm had access to a suitable rotation of the feature space, or to suitable projections of the features. Such transformations can be achieved by linear combinations of the features of the problem.

Such linear combinations are already available in virtually any SR system that allows for numeric constants and has the operators of addition and multiplication. However, these transformations have to be constructed using structural manipulation of the tree via mutation and crossover, and the values of the constants must be adjusted as well. Also, suitable transformations can be found only through the selection pressure. In this chapter we propose a new type of node for use in GP-based SR that explicitly provides a linear combination of features, and we show how it allows for informed tuning that does not rely solely on selection pressure.

This chapter is based on and extends article [C5].

5.1 Linear Combinations of Features

Linear Combinations of Features (LCFs) is an extension to GP-based SR systems that provides a new type of *leaf node*. LCFs perform linear combinations of input features and can be utilized by the GP algorithm as any other leaf node. Figure 5.1 illustrates how the model structure changes when LCFs are introduced.

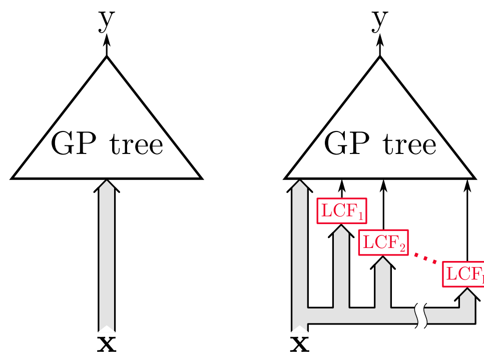


Figure 5.1. An illustration of a model structure without (left) and with (right) LCF nodes. The triangles denoted "GP tree" represent the inner structure of the model, **x** represents the feature vector and $LCF_1, LCF_2, \ldots, LCF_k$ represent the LCF leaf nodes.

Mathematically, the k-th LCF node implements the following function

$$LCF_k(\mathbf{x}; \mathbf{a}_k, b_k) = \mathbf{a}_k^\top \mathbf{x} + b_k = b_k + \sum_{i=1}^{N} a_{k,i} x_i \tag{5.1}$$

where $\mathbf{x}$ is the feature vector (vector of inputs to the model), N is its length, i.e. the number of features, and $\mathbf{a}_k$ and b_k are the LCF parameters. Similar extension was proposed for Single-Node Genetic Programming (SNGP) in [50]. In the rest of this chapter we are going to refer to the multiplicative constants $a_{k,i}$ and additive constant b_k simply as to *weights*.

The LCFs allow the underlying GP-based SR system to directly represent affine transformations of the feature space. To show that, let $f(\mathbf{x}) = f(x_1, \cdots, x_N)$ be a function of the original features. If these original features x_i are replaced by LCFs, the function changes to

$$\hat{f}(\mathbf{x}) = f(LCF_1(\mathbf{x}; \mathbf{a}_1, b_1), \cdots, LCF_N(\mathbf{x}; \mathbf{a}_N, b_N)). \tag{5.2}$$

Expanding the LCFs using Equation (5.1)

$$\hat{f}(\mathbf{x}) = f(\mathbf{a}_1^\top \mathbf{x} + b_1, \cdots, \mathbf{a}_N^\top \mathbf{x} + b_N). \tag{5.3}$$

Now, the individual vectors $\mathbf{a}_k$ can be rearranged into a matrix, the individual intercepts b_k can be rearranged into a vector

$$\hat{f}(\mathbf{x}) = f\left(\begin{pmatrix} \mathbf{a}_1^\top \\ \vdots \\ \mathbf{a}_N^\top \end{pmatrix} \mathbf{x} + \mathbf{b}\right) = f(\mathbf{A}\mathbf{x} + \mathbf{b}). \tag{5.4}$$

Obviously, the mapping $\mathbf{x} \mapsto \mathbf{A}\mathbf{x} + \mathbf{b}$ can represent rotation, translation, or scaling as its special cases.

The LCFs are not meant to replace all the features. Instead, an LCF is just another type of leaf node that is available to the GP algorithm to build trees from. A particular GP tree may contain no LCFs at all, but it is also possible that all leaf nodes of the tree will be LCFs. The original features are still available so the trees can use both the transformed and the original features. These features then enter the evolved tree at the bottom level and can be further transformed in non-linear ways by the rest of the nodes in the tree (see Figure 5.1). These transformations can be numerically optimized in an informed way using a gradient of the error w.r.t. the parameters of the transformations so that the whole function, including the non-linear parts, fits the target better.

We argue that LCFs are beneficial

1. in static scenarios when the target function is effectively a (potentially simple) function operating on a space transformed by a (close to) affine transformation, e.g. rotated (discussed in this chapter), or
2. in dynamic scenarios when the target changes between stages in a similar way (discussed in Chapter 6). The GP algorithm then does not have to deal with this change by creating a possibly much more complicated function, but it can adjust the coefficients of LCFs and possibly small parts of the tree, especially when the changes are gradual.

5.1.1 Initializing the LCF weights

Whenever a new LCF node is created, i.e. during generation of initial population or when a new subtree is generated due to a mutation, its weights must be initialized as well. We have chosen an approach that is as close as possible to not using any LCFs at all: when a new LCF node is created, one of the multiplicative weights, $a_{k,i}$, corresponding to certain input variable x_i (which is chosen randomly) is set to 1, and all other weights, including b_k, are set to 0. With this initialization procedure, a freshly created LCF node behaves exactly as a single (randomly chosen) variable.

There are, however, other possible initialization procedures. An obvious one is to just sample the parameters randomly from a predefined distribution, or to set all parameters equally to some predefined constant.

5.1.2 Tuning the weights

We designed three ways of tuning the weights:

- using a gaussian mutation, i.e. adding a normally distributed random number to a weight, and let the evolution choose better settings,
- using an informed gradient descent-based approach, and
- combination of the two previous approaches, i.e. using gradient-based tuning with an occasional perturbation of the weights using a mutation.

The gradient-based tuning takes advantage of the fact that the task being solved is a regression, i.e. the features, the target values, as well as the error (fitness) function are fully known and available (in contrast to some of other tasks usually solved by GP). We use the backpropagation algorithm commonly used in neural networks [51][1] to determine the partial derivatives of the model error with respect to the weights. Similar approach was used in [52–54] where gradient descent was used to optimize values of constant leaf nodes, and in [55] to optimize "inclusion factors" which are weights assigned to inputs of addition and multiplication operators.

Once the partial derivatives are computed, an update algorithm can be used to modify these weights to decrease the error. We have chosen iRprop$^-$[56] for this task because of two reasons:

1. it is simple (both computationally and from the implementation point of view) yet efficient [57][2], and
2. it is numerically robust as it operates only with the signs of the derivatives rather than with the values or magnitudes.

Especially the second reason is important because there is generally no constraint on the inner structure of the trees in the GP environment. The gradients of the expressions can easily explode, producing derivatives that exceed the range of floating-point number representation.

In our implementation, the gradient-based tuning takes place just before an individual is to be evaluated for fitness. It is effectively a part of the fitness evaluation process since the output values which are the basis for fitness computation are required to compute the gradient. The tuning is composed of three steps:

[1] The backpropagation algorithm used in this thesis slightly differs from the classic backpropagation algorithm in that there can be also binary functions (e.g. multiplication) instead of only unary ones as in the case of neural networks.

[2] Although the reference shows that iRprop$^+$ is superior to iRprop$^-$, our preliminary experiments have shown that in the GP environment the latter variant performs better and with less overfitting, hence we decided to use it.

1. forward pass, i.e. evaluating the individual on the training data,
2. backward pass, i.e. propagating the error backwards all the way down to the LCF nodes and computing the partial derivatives of the error, and
3. update step, i.e. modifying the weights based on the partial derivatives computed in step 2.

These three steps are repeated a number of times which is a parameter of the algorithm.

5.1.3 Operation modes of LCFs

A single LCF node represents a single row of some linear transformation matrix and a single element of the offset vector (see Equations (5.2), (5.3) and (5.4)), i.e. a single element of an affine transformation. If there are more LFC nodes than there are features, it necessarily follows that the transformation matrix encoded in those LCF nodes no longer has linearly independent rows. However, it might be desirable to restrict the algorithm to search using only an independent set of LCF nodes. To accommodate this, we designed three operation modes of LCFs.

Unsynchronized mode. In this mode, there are no restriction on the LCF nodes at all. There can be any number of LCF nodes present in an individual.

Synchronized mode. In this mode, each LCF node created (both during initial population initialization or during new subtree creation in subtree mutation), it is randomly assigned an index between (and including) 1 and the number of features of the data. Then, all LCFs with the same index are forced to have the same weights. This way, effectively only a single affine transformation will be introduced by the LCF nodes (or only some of its dimensions since there can be less LCF nodes than is the dimensionality of the data).

From technical point of view, all LCF nodes are still treated independently. However, to get the synchronised behaviour, special handling is employed:

- After each backpropagation phase (if tuned by gradient-based approach), the computed values of partial derivatives are summed up among the LCF nodes with the same index and these summed values replace the ones in the individual LCF nodes. This way, all partial derivatives are equal among the same-indexed groups of LCF nodes and are therefore updated in the same way.
- If a “conflict” is detected, i.e. that an individual has multiple sets of LCF weights for some of the indexes (e.g. due to crossover or subtree mutation), all sets are evaluated and the best performing set is selected.

Globally synchronized mode. This mode is similar to the synchronized mode but the index-based grouping encompasses the whole population instead of single individuals. The motivation behind this mode is that should there truly be a globally suitable transformation of the feature space, the models can work together to find it.

Since there is only a single set of LCF nodes, using only mutation to tune them no longer makes sense because there is no population of LCF sets the selection could pick from and hence mutation would only be a random walk. Therefore, we always use the gradient-based approach, alone or accompanied by the mutation as means to escape possible local optima.

5.2 Related work

Gradient-based tuning of parameters has been proposed and used in several previous research works. In [52–54] gradient descent was used to optimize values of constant leaf nodes. In [55], so called “inclusion factors” are introduced, which are multiplicative constants from the interval $[0,1]$ that multiply the output of children of addition and multiplication nodes. The inclusion factors are also tuned using gradient descent-based approach.

In [25], gradient-based tuning of numeric parameters is necessary as the algorithm is deterministic. Such tuning is also standard practice in [27, 29].

The most similar approach to ours is in [50], where the authors proposed linear combinations of features in Single-Node Genetic Programming (SNGP) [58–59]. The authors introduce a new segment of the linearly arranged population of nodes where each node in this segment (size of which is a parameter; the authors use length equal to the number of features) is a tuple $v = \langle w_0, c_0, x_0, \cdots, w_N, c_N, x_N \rangle$ where N is the number of features, $w_j \in \mathbb{R}$, c_j is a pointer to a constant-valued node in the population, and x_j is the j-th original feature with $x_0 = 1$ to allow for the offset of the transformation. The value of such node is calculated as

$$v = \sum_{j=0}^{N} w_j o(c_j) x_j,$$

where $o(c_j)$ is the output of the node referenced by c_j. Tuning of this transformation is performed (1) through standard mutation operator by changing the reference c_j, and (2) through dedicated tuning, step which is a simple local search, where at each iteration of this search a random transformed variable is chosen and, with equal probability, either c_j is changed or w_j is modified by adding a normally distributed random number. If such change does not decrease the performance of the model it is accepted. The number of iterations of this process is a parameter of the algorithm. This approach is very similar to our synchronized mode with a local search-based tuning procedure, not utilizing the error gradient.

5.3 LCFs in MGGP

So far, we have described how LCFs plug into vanilla GP algorithm. However, the LCFs can be plugged into other GP-based algorithms as well[3]. As we established in previous chapter, we use Multi-Gene Genetic Programming (MGGP) instead of regular GP algorithm. MGGP is itself an extension to the classic GP aimed at the SR task and therefore the introduction of a new type of node is seamless.

In MGGP, each individual is composed of multiple, independent expression trees, called genes. The final model that an individual represents is then created by making a linear combination of the genes, including an intercept term. Formally, the function represented by an MGGP individual is

$$f(\mathbf{x}) = c_0 + c_1 g_1(\mathbf{x}) + c_2 g_2(\mathbf{x}) + \cdots + c_m g_m(\mathbf{x})$$

where m is the number of genes in the given individual, g_i are genes of the given individual, and c_i are constants of the linear combination. For the rest of this work, we refer to

[3] For a short discussion of the applicability of LCFs to other types of SR systems, see Section 5.3.2.

this linear combination of individual's genes as to the *top-level linear combination* as it is the final operation that is performed with the genes before providing an output value. The coefficients of the top-level linear combination are computed deterministically by using multiple linear regression on the outputs of the particular genes.

When the LCFs are plugged in the MGGP, individuals are composed of:

- LCF nodes, that add new features to the set of inputs, and allow the model to encode affine transformations of the input space.
- Several GP trees – genes – that allow to build non-linear functions, using the original variables and/or their affine transformations created by LCFs as inputs.
- Top-level linear combination that scales and shifts the outputs of the genes to best fit the target values.

Comparison of a general model structure of an MGGP individual without and with LCFs can be seen in Figure 5.2. It is similar to the vanilla GP case (see Figure 5.1) except there are multiple trees per individual and a linear regression that combines them into the single output.

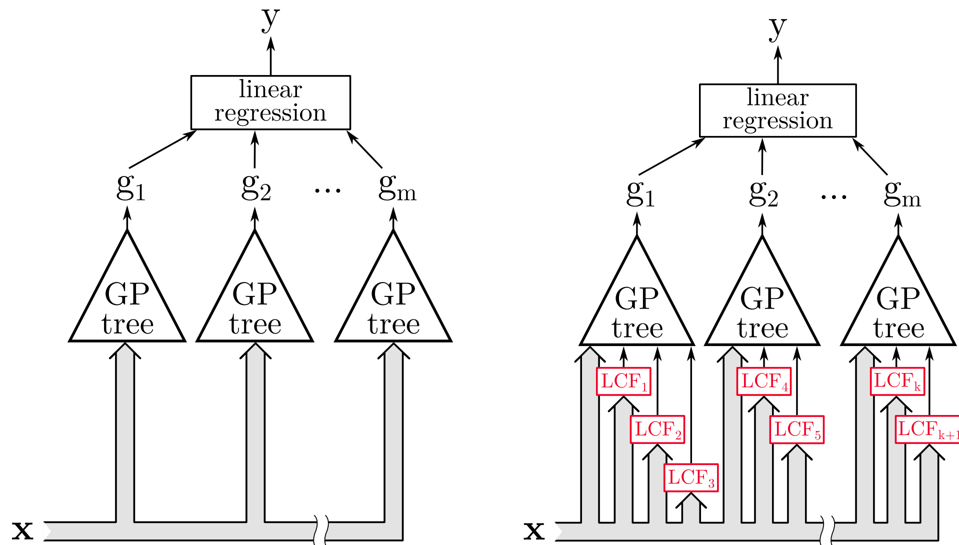


Figure 5.2. An illustration of a model structure represented by an MGGP individual without (left) and with (right) LCF nodes. The triangles denoted "GP tree" represent the inner structure of the genes, **x** represents the feature vector and $LCF_1, LCF_2, \ldots, LCF_k$ represent the LCF leaf nodes.

5.3.1 Genetic operators

There are two kinds of crossover operators used in MGGP[4]. The first one is the classic *subtree crossover*, which is identical to the one of vanilla GP, i.e two subtrees, one in each of the individuals being crossed over, are randomly chosen and swapped. The other kind of crossover operator is *high-level crossover*. High-level crossover swaps whole genes between the two individuals. These two operators are chosen stochastically whenever a crossover event takes place.

There are two mutation operators (which are not unique to MGGP): the classic subtree mutation which modifies the structure, and gaussian mutation which modifies the

[4] Though we have our own implementation of MGGP [60], it is identical (in terms of genetic operators of both the crossover and mutation) to Searson's GPTIPS 2 package [33] and we describe those crossover operators here.

values of constant leaf nodes. This Gaussian mutation is independent of the mutation used to perturb the LCF parameters. This means that whenever a mutation event takes place, one of the following mutations is chosen stochastically:

- subtree mutation,
- Gaussian mutation of constant leaf nodes, or
- Gaussian mutation of LCF nodes (if this mutation is used).

5.3.2 Incorporating LCFs in other SR algorithms

In principle, LCFs can be used in any SR system as it is just a subexpression to be used in the produced models. However, for various SR systems this can be more or less complicated. Clearly, if gradient-based tuning is to be used, the gradient needs to be computed which the system might not be ready for, and may need major implementation changes. However, LCFs can also be tuned using only mutation which is arguably easier to implement, though it does not use the gradient information and relies only on the selection pressure.

Another significant limitation might be the design of the algorithm. Some algorithms cache the outputs of subexpressions to speed up the computation.[5] This assumes that values of subexpressions stay constant once they are evaluated. However, the tuning of LCFs causes their outputs to change (including the related parts of GP trees). Therefore algorithms that take advantage of caching would need to be modified (possibly requiring a major redesign) to allow for changing subexpressions, which might lead to the algorithm being slowed down considerably.

5.4 Research questions

In the rest of this chapter, we aim to answer the following research questions:

- RQ1: Are LCFs beneficial when an affine transformation is truly present in the data?
- RQ2: Are LCFs beneficial in other cases?
- RQ3: Are LCFs with gradient-based tuning better than using the same kind of tuning on numeric constants?
- RQ4: Which algorithm configuration with LCFs should be adopted as the best one, or the default, if any?

5.5 Experimental evaluation

In this section we experimentally examine the viability of LCFs in the context of the research questions we have stated in the previous section. We perform a series of experiments in the classic regression task: given a training data and a fixed amount of time, find a model that fits the target variable as well as possible also on unseen data from the same distribution.

We first evaluate our approach on a set of toy problems, specifically designed in a way that LCFs should be beneficial. These should answer RQ1. We then move to a more realistic set of problems which is composed of two artificial ones, four real-world ones,

[5] An example of such algorithm is EFS [15], which does not actually hold the trees at all (it only logs them in order to be able to retrieve the final model) and it only works with the outputs of the (sub)trees.

and three problems that are related to Reinforcement Learning (RL). These should answer RQ2. RQ3 and RQ4 are going to be answered by comparing the performance of carefully chosen algorithm configurations, description of which follows.

5.5.1 Algorithm configurations

The base algorithm to which the LCFs are added as an extension is Multi-Gene Genetic Programming (MGGP) algorithm [10] which shall also serve as a baseline for comparison. In Section 5.1.2 we proposed three methods of tuning the LCF weights, and in Section 5.1.3 we proposed three operation modes. We shall examine all sensible combinations of these[6]. These algorithm configurations shall be identified by 2-letter "codenames" with the first letter signifying the operation mode and the second one the method of tuning the weights. These code letters are summarised in Table 5.1.

	code letter	description
operation mode	U	unsynchronised
	S	synchronised
	G	globally synchronised
tuning method	M	mutation
	B	gradient-based
	C	combined M and B

Table 5.1. Description of codename letters of the algorithm configurations.

In order to answer the research questions, two extra configurations are added to the comparison as well. The first one is the MGGP algorithm without LCFs. It shall have no codename assigned and will be referred to simply as MGGP. The second extra configuration, introduced specifically to answer RQ3, is MGGP without LCFs but with numerical constants tuned using the same gradient-based approach as for the LCF parameters. It will be referred to as MGGP+TC (TC standing for Tuned Constants). These two extra configurations should serve as baselines. Comparison with MGGP should show whether LCFs bring an improvement, while comparison with MGGP+TC should show whether it is the explicit linear combinations that make the difference (if any), or rather the gradient-based tuning itself.

Note. MGGP+TC is very similar to the approach proposed in [52, 54]. In both of these works, numeric leaf nodes were tuned using a gradient descent. The main differences are (1) we use MGGP as the base algorithm (they used GP), and (2) we use the iRprop$^-$ update rule while [52] used simple gradient descent with adaptive learning rate and [54] used the Levenberg-Marquardt algorithm.

5.5.2 Algorithm parameters

The introduction of LCFs naturally comes with a number of parameters that exactly specify the behaviour of the algorithm. Table 5.2 summarises them together with those that come from the base algorithm of MGGP. The values of MGGP parameters are based on the defaults provided by the GPTIPS2 [33] package with a few exceptions, namely G_{max}, D_{max} and the function set. The values of LCF-related parameters were chosen manually based on preliminary experiments. All MGGP parameters are the

[6] As we argued in 5.1.3, we don't consider globally synchronised mode with mutation-only tuning as sensible and therefore we omit this combination.

same across all configurations with the exception of configurations using gradient-based tuning where we decided to halve the population size because more time is spent with each individual due to the tuning and there would be little time left for structural exploration.

	param.	description	value
MGGP	G_{max}	maximum number of genes	10
	N_{max}	maximum number of nodes	∞
	D_{max}	maximum depth of a gene	11
	$\lvert P\rvert$	population size	100 (50)
	$\lvert T\rvert$	tournament size	10 (5)
	E	number of top individuals directly copied to next gen. (elitism)	15 (8)
	Pr_x	probability of crossover	0.84
	Pr_m	probability of mutation	0.14
	Pr_{HLx}	probability of high-level crossover (given a crossover takes place)	0.2
	Pr_{LLx}	probability of low-level crossover (given a crossover takes place)	$1 - Pr_{HLx}$
	r_{HLx}	probability of a gene being selected for high-level crossover	0.5
	Pr_{Sm}	probability of subtree mutation (given a mutation takes place)	$1 - Pr_{Cm} - Pr_{Wm}$
	Pr_{Cm}	probability of constant leaf node mutation	0.05
	σ_{Cm}	variance of the gaussian distribution for constant leaf node mutation	0.1
LCF	Pr_{Wm}	probability of LCF weights mutation	0.05
	σ_{Wm}	variance of the gaussian distribution for LCF weights mutation	3
	Bp_{steps}	base number[a] of backprop.+update steps	25
	Bp_{min}	minimum number[a] of backprop.+update steps	2

[a] The actual number of backprop.+update steps is determined as $\max\{Bp_{min}, Bp_{steps} - \#nodes\}$.

Table 5.2. Algorithm parameters. Values in parentheses indicate the values used in configurations which employ gradient-based tuning, i.e. configurations UB, UC, SB, SC, GB, GC, and MGGP+TC.

Function set (non-terminal nodes) available to the algorithm were (x and y being placeholders for child nodes): $x + y$, $x - y$, xy, $\sin x$, $\cos x$, e^x, $\frac{1}{1+\mathrm{e}^{-x}}$, $\tanh x$, $\frac{\sin x}{x}$, $\ln(1 + \mathrm{e}^x)$, e^{-x^2}, x^2, x^3, x^4, x^5, and x^6.

5.5.3 Testing methodology and environment

For each algorithm configuration and benchmark, 30 runs were computed, each with a different training/testing sampling. Each run was limited to 7 minutes of wall-clock time. Fitness is R^2 on training data and is maximized. After a run is finished, the final model (the one with best training fitness) is evaluated on the testing set.

All tests were carried out on the National Grid Infrastructure MetaCentrum (see Acknowledgements). We ensured that all runs were performed on nodes of identical configuration[7].

[7] 2x 8-core Intel Xeon E5-2650 v2 2.6 GHz, Debian 8.7 (Jessie), SPECfp2006 [61] score 490 (30.6 per core). However, the algorithms are single-threaded so only one core was utilized.

5.5.4 Toy problems

We provide two toy problems, called S5D and RS5D. The function that produces the S5D problem is defined as

$$S(\mathbf{x}) = \frac{1}{1 + e^{-x_1}} \tag{5.5}$$

where $\mathbf{x}$ is 5-dimensional. In other words, it is a sigmoid function applied to the first dimension; the target is independent of the other dimensions.

The function that produces the RS5D problem is defined as

$$RS(\mathbf{x}) = S(\mathbf{Rx}) \tag{5.6}$$

where $\mathbf{x}$ is 5-dimensional and $\mathbf{R}$ is a rotation matrix that rotates the feature vector by $\frac{\pi}{4}$ in all pairs of axes. Therefore, in the RS5D problem the target value depends on all features.

Both training and testing data is uniformly randomly sampled from the interval $[-10, 10]^5$, there are 500 samples in the training set and 1250 samples in the testing set.

The expectation is following:

- All configurations should perform well on the S5D problem. The target function is very easy because the sigmoid function is among the functions available to the algorithm (see Section 5.5.2).
- On the RS5D problem, the algorithm with LCFs should perform significantly better than without them because LCFs provide means to do the rotation directly, without the need to arrange the coefficients by manipulating the tree structure.

We have no expectations regarding the performance of the three different operation modes as well as the base-TC configuration.

5.5.5 Results on toy problems

The results are presented in Tables 5.3 and 5.4. For both tables, the first two rows show the performance of the base (pure MGGP) and base-TC (MGGP with tunable constants) configurations, the columns "vs. base" and "vs. base-TC" indicate whether the corresponding configuration was better than (denoted by ✓), worse than (denoted by ×) or indifferent to (denoted by blank space) the base and base-TC configuration respectively, which was established using the MWUT on the testing R^2 with the significance level $\alpha = 0.01$. Columns "mean LCF", "mean consts" and "mean depth" provide a high-level view on the overall structure of the resulting models by showing the mean proportion of LCF nodes among non-constant leaf nodes (i.e. $\frac{\#\text{LCFs}}{\#\text{LCFs} + \#\text{original features}}$), mean proportion of constant leaf nodes among all leaf nodes and mean maximum depth of the models respectively.

Discussion. The first expectation – that all configurations work well on the unrotated problem – was confirmed, as can be clearly seen in Table 5.3. The second expectation – that the rotated problem is easy for configurations with LCFs but not for the base – was confirmed as well, though not for all configurations.

In Table 5.4 it can be seen that UM and SM were not significantly better or worse than MGGP or MGGP+TC and GC was significantly worse than MGGP. Those configurations also show smaller ratios of LCFs than the other configurations. For UM and SM, this indicates that tuning the weights only by mutation is not enough and thus the LCFs are not used as much and the performance is similar to MGGP(+TC). The

config.	training R^2		testing R^2		vs. MGGP	vs. MGGP+TC	mean LCF	mean consts	mean depth
	median	max min	median	max min					
MGGP	1	1 1	1	1 1			–	0.161	4.33
MGGP+TC	1	1 1	1	1 1			–	0.157	4.40
UM	1	1 1	1	1 1			0.482	0.061	4.10
UB	1	1 1	1	1 1			0.576	0.053	3.50
UC	1	1 >0.999	1	1 >0.999			0.521	0.056	3.77
SM	1	1 1	1	1 1			0.469	0.054	4.13
SB	1	1 1	1	1 1			0.475	0.052	4.20
SC	1	1 >0.999	1	1 >0.999			0.480	0.054	3.73
GB	1	1 1	1	1 1			0.380	0.076	4.23
GC	1	1 >0.999	1	1 >0.999			0.352	0.073	4.87

Table 5.3. Results on the S5D toy problem. The columns "vs. MGGP" and "vs. MGGP+TC" mark whether the configuration was significantly better or worse than MGGP and MGGP+TC respectively, as established by MWUT. The column "mean LCF" shows the mean proportion of LCF nodes among non-constant leaf nodes. The column "mean consts" shows the mean proportion of numeric leaf nodes among all leaf nodes.

config.	training R^2		testing R^2		vs. MGGP	vs. MGGP+TC	mean LCF	mean consts	mean depth
	median	max min	median	max min					
MGGP	0.995	0.997 0.942	0.991	0.996 0.912			–	0.016	10.90
MGGP+TC	0.990	0.997 0.861	0.987	0.995 0.215			–	0.061	10.90
UM	0.993	>0.999 0.930	0.990	>0.999 0.907			0.524	0.029	10.63
UB	>0.999	1 >0.999	>0.999	1 >0.999	✓	✓	0.980	0.000	4.60
UC	>0.999	1 >0.999	>0.999	1 >0.999	✓	✓	0.974	0.000	4.63
SM	0.995	>0.999 0.158	0.993	>0.999 -0.168			0.579	0.031	10.90
SB	>0.999	>0.999 >0.999	>0.999	>0.999 >0.999	✓	✓	0.900	0.001	8.10
SC	>0.999	>0.999 >0.999	>0.999	>0.999 >0.999	✓	✓	0.954	0.005	7.77
GB	>0.999	>0.999 0.855	>0.999	>0.999 0.659	✓	✓	0.817	0.008	8.53
GC	0.974	>0.999 0.872	0.962	>0.999 0.736	×		0.651	0.032	6.70

Table 5.4. Results on the RS5D toy problem. The columns "vs. MGGP" and "vs. MGGP+TC" mark whether the configuration was significantly better or worse than MGGP and MGGP+TC respectively, as established by MWUT. The column "mean LCF" shows the mean proportion of LCF nodes among non-constant leaf nodes. The column "mean consts" shows the mean proportion of numeric leaf nodes among all leaf nodes.

case of GC is interesting when compared to GB. Both GB and GC use backpropagation to tune the weights but GC also mutates them. GB was able to beat both base and base-TC while GC was beaten by base and indifferent to base-TC. This indicates that in the globally synchronised mode, the random perturbations are detrimental to the

performance of the algorithm. A possible reason for these results of GC is that because of the global synchronisation, there is no “population” of LCFs to choose from and therefore bad mutations cannot be subject to selection pressure.

We can also see that the successful configuration have high LCF ratio which indicates that the LCFs are really being used. Another interesting aspect is the depth usage: MGGP, MGGP+TC, UM and SM configurations utilise almost all the available depth (limit was 11) while SB, SC and GB utilise only about 8 levels of depth, and UB and UC even use only about 4.6 levels on average. This is an indication that the LCFs provide good projections that don’t need to be manipulated much further.

Regarding the performance of MGGP+TC, it was shown indifferent to MGGP and other configurations are in almost identical relationship to it as they are to MGGP. This indicates that in this case, gradient-based tuning of constants neither improves nor degrades the performance of the pure MGGP algorithm.

5.5.6 Realistic problems

There are 9 problems that we consider “realistic” because of the number of samples, dimensions, their real-world nature, or a combination of these criteria.

K11C is an artificial dataset, a modified version of Keijzer11 from [34]. The original Keijzer11 is defined as

$$f(x_1, x_2) = x_1 x_2 + \sin\left(\left(x_1 - 1\right)\left(x_2 - 1\right)\right)$$

while our modified version adds several numerical constants throughout the formula and is defined as

$$f(x_1, x_2) = \left(27.22 x_1 - 4.54\right)\left(-0.39 x_2\right) + 11.46 \sin\left(\left(0.21 x_1 - 1\right)\left(x_2 + 16.6\right) + 1.97\right)$$

We introduced the modification so that the algorithms cannot make a good model using only the function and variable nodes. The training set is 500 uniformly randomly sampled datapoints from the range $[-3, 3]^2$. The testing set is a grid from the same range with spacing of 0.01 in each dimension, resulting in 361201 datapoints.

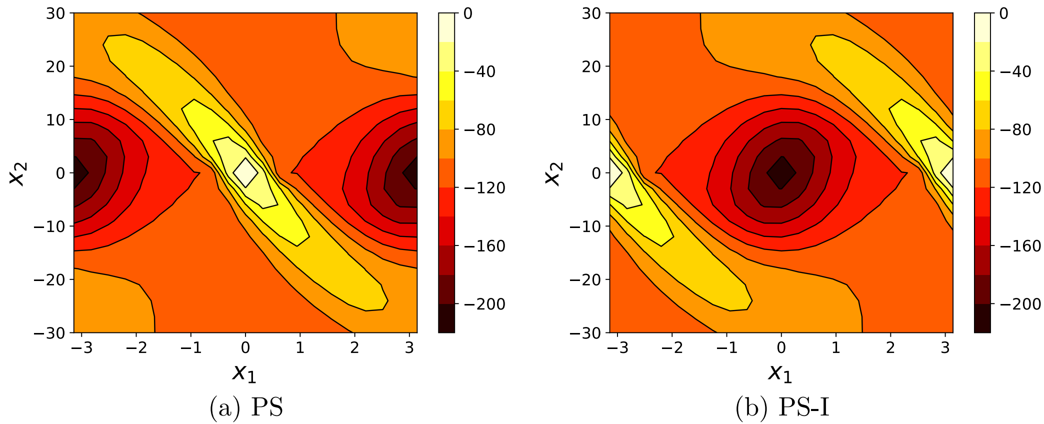
UB5D (Unwrapped Ball 5D) [36] is also an artificial dataset. It is defined as

$$f(\mathbf{x}) = \frac{10}{5 + \sum_{i=1}^{N} (x_i - 3)^2}$$

where $N = 5$. The training and testing sets are 1024 and 5000 uniformly randomly sampled from the range $[-0.25, 6.35]^5$.

ASN, CCS, ENC, ENH are four real-world datasets that we have already used in the previous chapter and we use them here as well, in exactly the same way. See Section 4.3.2 for details.

PS, PS-I (pendulum swingup) are two datasets from a reinforcement learning (RL) domain. They represent value functions of an inverted pendulum swing-up problem, computed by a numeric approximator. They are 2-dimensional (pendulum angle and angular velocity) and the value for a given point is the value of the state w.r.t. the target position which, for the PS variant, is located at $[0, 0]$. The PS-I variant is identical except the first coordinate (pendulum angle) is shifted by $\frac{\pi}{2}$ so the target position is at $[\pm\pi, 0]$ (due to the circular nature of the problem). There are 441 samples which are



(a) PS (b) PS-I

Figure 5.3. Depiction of the PS and PS-I datasets. Note that the non-smoothness of the contour lines is an artifact of the display method and the fact, that we do not have the actual function available and therefore we display the data. An optimal V-function (which this data approximates), if it was known, would be smooth [62–63]

randomly split into training and testing sets in the proportion 70:30. Both datasets are depicted in Figure 5.3.

MM (magnetic manipulation) is another RL-based dataset. It represents a value function of a 2-coil linear magnetic manipulator that manipulates a steel ball's position on a linear track, again computed by a numeric approximator. It is also 2-dimensional (the ball's position and velocity). There are 729 samples randomly split into training and testing sets in the proportion 70:30. The dataset is depicted in Figure 5.4.

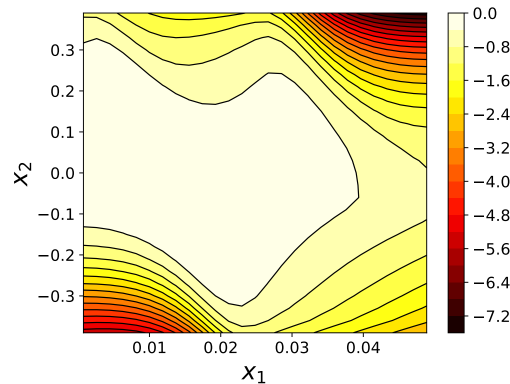


Figure 5.4. Depiction of the MM dataset.

5.5.7 Results on realistic problems

First, we present a summary of which configurations were better than, indifferent to or worse than the MGGP and MGGP+TC configurations, which can be seen in Table 5.5. The table shows how a particular configuration was compared to the baseline configurations across all the datasets, using the same statistical test and its settings as for the "vs. MGGP" and "vs. MGGP+TC" columns in the tables in previous subsection.

First of all, we can see that MGGP+TC did not outperform MGGP on any dataset and was outperformed by MGGP on three of them. Next, we can see that UM and SM were, except for a single case, indifferent to MGGP and outperformed MGGP+TC

config.	MGGP+TC	UM	UB	UC	SM	SB	SC	GB	GC
better than MGGP		1	5	5	1	3	3		
indifferent to MGGP	6	8	3	3	8	4	4	6	3
worse than MGGP	3		1	1		2	2	3	6
better than MGGP+TC	–	4	5	6	3	5	4		
indifferent to MGGP+TC	–	5	4	3	6	4	5	7	6
worse than MGGP+TC	–							2	3

Table 5.5. Summary results for each configuration on the realistic problems. The numbers show on how many problems was the particular configuration better than, indifferent to, or worse than the baseline.

on 4 and 3 datasets respectively. Also, GB and GC were never better than neither MGGP nor MGGP+TC.

The results for individual datasets are presented in Tables 5.6 through 5.14 and Figures 5.5 through 5.13 which show box plots of the final values.

Discussion. In the summary results (see Table 5.5) we can see that the configurations using globally synchronised mode were in no case better than either of the baselines. We hypothesise that this is caused by the fact that both structurally good and structurally bad individuals can be present in the population. These, in turn, "fight" each other when the LCF parameters are to be updated, stalling the progress. The usefulness of the globally synchronised mode is therefore doubtful.

We can also see that the (locally) synchronised mode does not perform as well as the unsynchronized mode. We hypothesise that this is caused by the need to resolve conflicts when structural changes (i.e. subtree mutation or crossover) take place (see Section 5.1.3; synchronized mode). After a conflict is resolved, some LCFs necessarily have their weights changed from what they were before the structural change happened which were presumably optimised. These optimised weights are then lost due to the conflict resolution. Also, this extra step requires evaluation of the individual multiple times (once for each set of conflicting sets of LCFs) which takes some time.

Another clearly visible result is that using only mutation for tuning the weights makes almost no difference compared to MGGP which is in line with the results on the toy problems (see Section 5.5.5). However, the configurations tuned by gradient descent (except for GB which we already discussed and MGGP+TC which we discuss below) were able to improve on the baseline performance of MGGP, especially UB and UC. This indicates that this technique is a viable approach to tune the LCFs.

The MGGP+TC configuration, i.e. MGGP where the numerical constants are tuned in the same way as are the LCF parameters in LCF-enabled configurations, did not show an improvement over MGGP. Also, all the LCF-enabled configurations compared better against MGGP+TC than against MGGP. At first glance, this seems to be in contradiction to [52] and [54] which both tested the idea of gradient-based tuning of numerical constants. Although their algorithms are similar in principle, there are significant differences to our setup. The one we deem the most important to the underperformance of MGGP+TC is that the base algorithm is MGGP rather than GP that was used in both works. We think that the effect of having multiple genes combined optimally by top-level linear combination is itself so powerful that it already greatly improves the performance when compared to ordinary GP. Optimization of the numerical constants then introduces extra computational cost that degrades the performance of the algorithm. This, however, does not happen when the optimized constants are bound to

config.	training R^2 median	training R^2 max / min	testing R^2 median	testing R^2 max / min	vs. MGGP	vs. MGGP+TC	mean LCF	mean consts	mean depth
MGGP	0.981	0.997 / 0.971	0.976	0.995 / 0.965		✓	–	0.070	11.00
MGGP+TC	0.976	0.994 / 0.963	0.970	0.991 / 0.956	×		–	0.182	10.97
UM	0.986	0.997 / 0.975	0.981	0.996 / 0.969	✓	✓	0.542	0.072	10.97
UB	0.998	>0.999 / 0.990	0.996	>0.999 / 0.978	✓	✓	0.873	0.038	8.87
UC	0.998	>0.999 / 0.992	0.997	>0.999 / -3.2e29	✓	✓	0.874	0.024	8.53
SM	0.986	0.997 / 0.973	0.981	0.997 / 0.966	✓	✓	0.595	0.068	11.00
SB	0.991	0.998 / 0.954	0.989	0.998 / 0.945	✓	✓	0.603	0.032	10.13
SC	0.992	0.998 / 0.954	0.990	0.997 / 0.948	✓	✓	0.622	0.026	9.93
GB	0.972	0.993 / 0.958	0.967	0.992 / 0.952			0.549	0.028	10.53
GC	0.971	0.989 / 0.956	0.966	0.987 / 0.949	×		0.177	0.085	10.40

Table 5.6. Results on the K11C problem. Columns “vs. MGGP” and “vs. MGGP+TC” mark whether the configuration was significantly better, worse, or indifferent to the base and base-TC configurations based on MWUT. The column “mean LCF” shows the mean proportion of LCF nodes among non-constant leaf nodes. The column “mean consts” shows the mean proportion of constant leaf nodes among all leaf nodes.

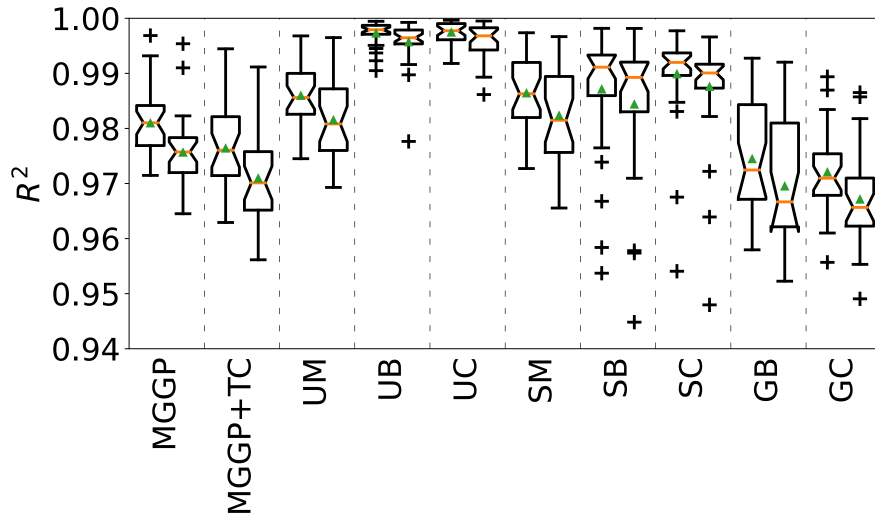


Figure 5.5. Training (left) and testing (right) R^2 boxplots over all 30 runs on the K11C dataset. Note that for UC, one test-set outlier at -3.24e29 is not shown.

the features, as is done in LCFs. This suggests that, for MGGP, not only the tuning is important, but also the structure of LCFs.

On the K11C dataset (see Table 5.6 and Figure 5.5) we can see a strong result in favour of UB and UC configurations. The true relationship contains several terms of the form $ax + b$ (see Section 5.5.6), which is exactly what LCFs are capable to express, so the result of the LCF-enabled configuration is a strong indication that LCFs themselves are beneficial.

config.	training R^2 median	training R^2 max	training R^2 min	testing R^2 median	testing R^2 max	testing R^2 min	vs. MGGP	vs. MGGP+TC	mean LCF	mean consts	mean depth
MGGP	0.885	0.976	0.808	0.866	0.968	0.796			–	0.032	11.00
MGGP+TC	0.851	0.972	0.818	0.835	0.966	0.800			–	0.111	10.60
UM	0.884	0.965	0.821	0.862	0.966	0.802			0.539	0.009	10.93
UB	0.857	0.887	0.828	0.828	0.856	0.580	×		0.823	0.009	5.33
UC	0.858	0.932	0.824	0.826	0.892	0.807	×		0.802	0.024	6.40
SM	0.907	0.982	0.813	0.890	0.977	0.805			0.533	0.015	10.87
SB	0.839	0.972	0.802	0.816	0.967	0.796	×		0.553	0.010	6.47
SC	0.839	0.930	0.816	0.818	0.908	0.795	×		0.601	0.016	6.13
GB	0.825	0.881	0.761	0.808	0.873	0.749	×	×	0.334	0.014	8.00
GC	0.828	0.902	0.778	0.808	0.887	0.783	×	×	0.068	0.013	8.13

Table 5.7. Results on the UB5D problem. Columns "vs. MGGP" and "vs. MGGP+TC" mark whether the configuration was significantly better, worse, or indifferent to the base and base-TC configurations based on MWUT. The column "mean LCF" shows the mean proportion of LCF nodes among non-constant leaf nodes. The column "mean consts" shows the mean proportion of constant leaf nodes among all leaf nodes.

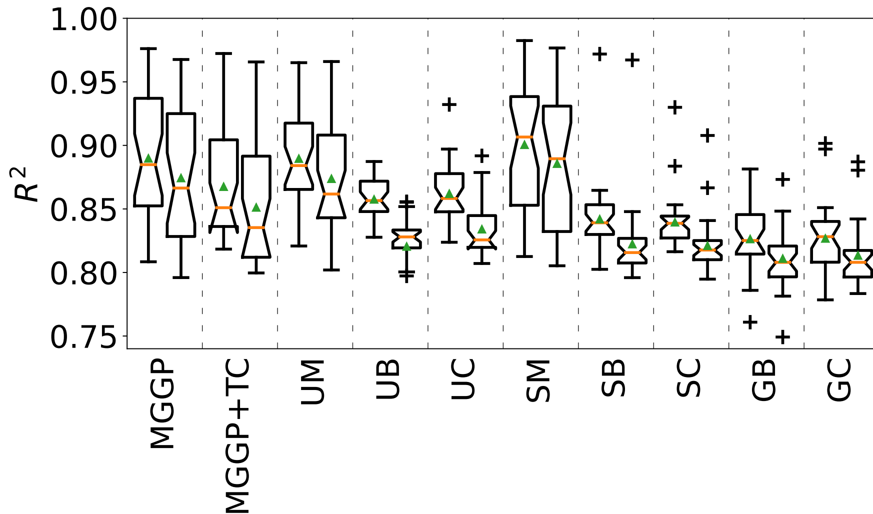


Figure 5.6. Training (left) and testing (right) R^2 boxplots over all 30 runs on the UB5D dataset. Note that for UB, one test-set outlier at 0.580 is not shown.

On the UB5D dataset (see Table 5.7 and Figure 5.6), the LCF-enabled configurations with gradient-based tuning showed worse behaviour than MGGP. We hypothesize this is caused by the fact that the true function is a single fraction and there is no division operator in the function set, because of which the potentially good optimization of the LCF parameters can push the models into local optima. Without the gradient-tuned LCRs (i.e. the MGGP, MGGP+TC, UM, and SM configurations), the algorithm doesn't waste time optimizing the parameters (except for occasional mutation for the

config.	training R^2		testing R^2		vs. MGGP	vs. MGGP+TC	mean LCF	mean consts	mean depth
	median	max / min	median	max / min					
MGGP	0.842	0.892 / 0.720	0.824	0.885 / 0.625		✓	–	0.063	11.00
MGGP+TC	0.808	0.868 / 0.719	0.783	0.855 / 0.658	×		–	0.089	11.00
UM	0.845	0.890 / 0.787	0.824	0.890 / 0.729		✓	0.461	0.021	10.93
UB	0.849	0.914 / 0.729	0.818	0.893 / -0.719			0.834	0.005	6.30
UC	0.841	0.894 / 0.705	0.818	0.880 / 0.623			0.828	0.001	5.77
SM	0.836	0.887 / -7.226	0.811	0.884 / -4.135		✓	0.462	0.020	11.00
SB	0.804	0.842 / 0.675	0.770	0.829 / 0.624	×		0.651	0.013	7.50
SC	0.800	0.867 / 0.710	0.760	0.861 / 0.653	×		0.680	0.011	7.43
GB	0.817	0.860 / 0.631	0.788	0.880 / 0.584			0.430	0.012	8.80
GC	0.778	0.849 / 0.669	0.757	0.859 / 0.645	×		0.309	0.023	8.47

Table 5.8. Results on the ASN problem. Columns "vs. MGGP" and "vs. MGGP+TC" mark whether the configuration was significantly better, worse, or indifferent to the base and base-TC configurations based on MWUT. The column "mean LCF" shows the mean proportion of LCF nodes among non-constant leaf nodes. The column "mean consts" shows the mean proportion of constant leaf nodes among all leaf nodes.

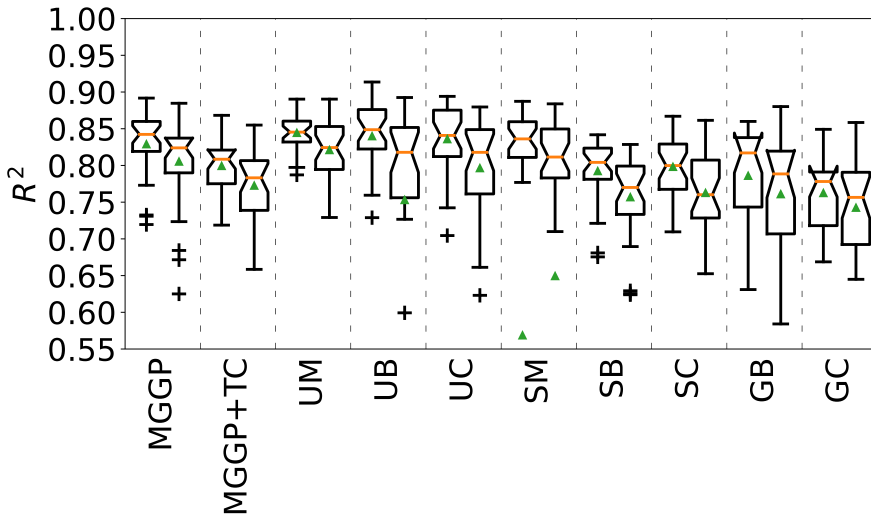


Figure 5.7. Training (left) and testing (right) R^2 boxplots over all 30 runs on the ASN dataset. Note that for UB, one test-set outlier at -0.719 is not shown, as well as two test-set outliers at -7.226 and -4.135 for SM.

UM and SM variants) and just tries to find a structure that fits well on the range of the data. Similar result can be observed on the ASN dataset (see Table 5.8 and Figure 5.7), although not as pronounced. Also, since ASN is a real-world dataset and we don't know the true underlying relationship, we cannot make the same hypothesis.

A fact worth pointing out is that the presented configurations, especially UB and UC, performed very well on the RL domain datasets (PS, PS-I, MM). There are common characteristics of these three datasets: low dimensionality (2 dimensions), the target

config.	training R^2		testing R^2		vs. MGGP	vs. MGGP+TC	mean LCF	mean consts	mean depth
	median	max min	median	max min					
MGGP	0.869	0.890 0.848	0.844	0.868 -8.68e7			–	0.042	10.97
MGGP+TC	0.856	0.874 0.840	0.832	0.865 -2.331			–	0.042	10.97
UM	0.866	0.885 0.851	0.839	0.865 -1.16e6			0.496	0.026	11.00
UB	0.901	0.924 0.869	0.859	0.892 0.806	✓	✓	0.870	0.000	4.77
UC	0.899	0.931 0.854	0.858	0.880 0.758	✓	✓	0.885	0.011	4.30
SM	0.862	0.885 0.846	0.837	0.882 0.799			0.467	0.026	11.00
SB	0.889	0.906 0.868	0.851	0.898 -4.74e4		✓	0.676	0.000	6.10
SC	0.893	0.908 0.857	0.846	0.873 -290.91			0.707	0.019	5.60
GB	0.859	0.885 0.844	0.825	0.867 -99.601			0.430	0.026	9.73
GC	0.854	0.867 0.836	0.830	0.879 -7.35e6			0.252	0.004	7.40

Table 5.9. Results on the CCS problem. Columns "vs. MGGP" and "vs. MGGP+TC" mark whether the configuration was significantly better, worse, or indifferent to the base and base-TC configurations based on MWUT. The column "mean LCF" shows the mean proportion of LCF nodes among non-constant leaf nodes. The column "mean consts" shows the mean proportion of constant leaf nodes among all leaf nodes.

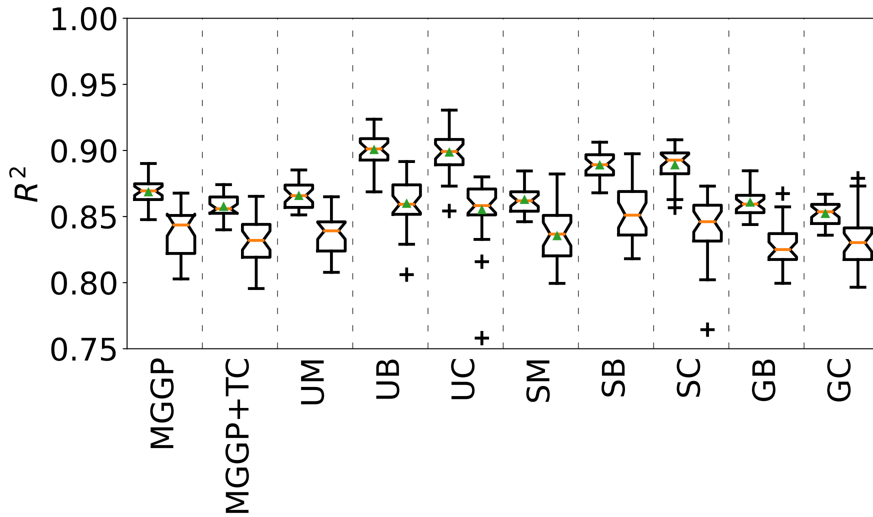


Figure 5.8. Training (left) and testing (right) R^2 boxplots over all 30 runs on the CCS dataset. Note that multiple outliers are not shown: at -86.81e6 for base, at -2.331 for base-TC, at -1.16e6 and 0.442 for UM, at -47.41e3 for SB, at -290.9 and 0.478 for SC, at -99.60 and -5.382 for GB, and at -7.35e6 for GC, all of them for test set.

shape is smooth, without sharp peaks or rapid oscillations, and there is no noise. It is possible that LCFs are well suited to these kinds of problems. This explanation would be supported also by the results on the toy problems and the K11C dataset as those, too, share these characteristics.

config.	training R^2			testing R^2			vs. MGGP	vs. MGGP+TC	mean LCF	mean consts	mean depth
	median	max	min	median	max	min					
MGGP	0.974	0.981	0.969	0.970	0.982	0.963			–	0.050	11.00
MGGP+TC	0.972	0.980	0.970	0.968	0.973	0.962			–	0.054	10.97
UM	0.974	0.984	0.971	0.970	0.980	0.961			0.548	0.031	10.97
UB	0.974	0.988	0.970	0.969	0.982	0.957			0.751	0.006	6.60
UC	0.975	0.986	0.972	0.971	0.985	0.961		✓	0.772	0.000	5.27
SM	0.974	0.985	0.970	0.969	0.979	0.960			0.520	0.009	10.90
SB	0.974	0.979	0.969	0.968	0.973	0.965			0.609	0.000	7.23
SC	0.973	0.980	0.969	0.968	0.976	0.962			0.609	0.003	7.27
GB	0.971	0.980	0.967	0.967	0.981	0.954	×		0.551	0.008	8.53
GC	0.971	0.973	0.968	0.967	0.972	0.957			0.123	0.008	8.50

Table 5.10. Results on the ENC problem. Columns "vs. MGGP" and "vs. MGGP+TC" mark whether the configuration was significantly better, worse, or indifferent to the base and base-TC configurations based on MWUT. The column "mean LCF" shows the mean proportion of LCF nodes among non-constant leaf nodes. The column "mean consts" shows the mean proportion of constant leaf nodes among all leaf nodes.

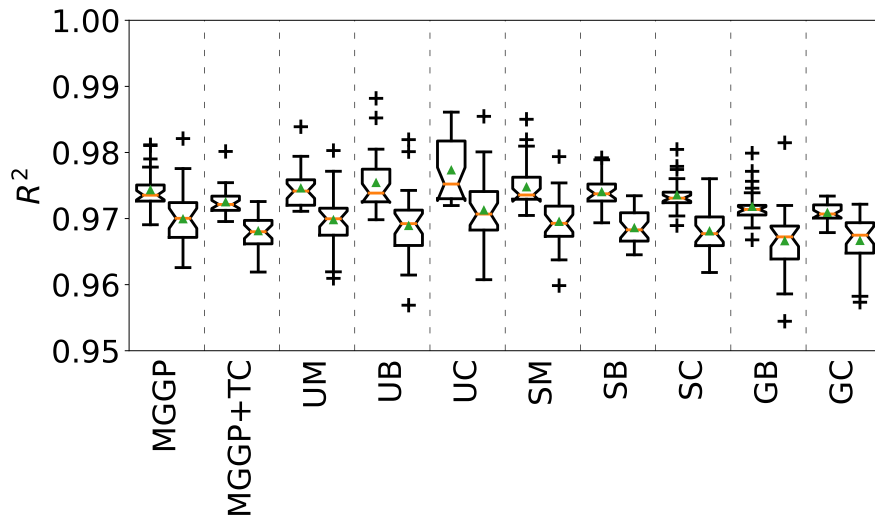


Figure 5.9. Training (left) and testing (right) R^2 boxplots over all 30 runs on the ENC dataset.

config.	training R^2		testing R^2		vs. MGGP	vs. MGGP+TC	mean LCF	mean consts	mean depth
	median	max min	median	max min					
MGGP	0.998	0.998 0.996	0.997	0.998 0.995		✓	–	0.030	11.00
MGGP+TC	0.997	0.998 0.994	0.996	0.998 0.994	×		–	0.063	10.97
UM	0.998	0.998 0.996	0.997	0.998 0.995		✓	0.501	0.010	10.93
UB	0.997	0.998 0.993	0.997	0.998 0.991			0.730	0.000	4.70
UC	0.998	0.998 0.995	0.997	0.998 0.994			0.732	0.003	5.90
SM	0.997	0.998 0.996	0.997	0.998 0.996		✓	0.546	0.032	10.90
SB	0.997	0.998 0.993	0.997	0.998 0.993			0.592	0.003	6.87
SC	0.997	0.998 0.990	0.997	0.998 0.988			0.610	0.000	6.97
GB	0.996	0.998 0.990	0.996	0.997 0.990	×	×	0.487	0.006	8.07
GC	0.996	0.998 0.989	0.996	0.997 0.986	×	×	0.099	0.005	8.90

Table 5.11. Results on the ENH problem. Columns "vs. MGGP" and "vs. MGGP+TC" mark whether the configuration was significantly better, worse, or indifferent to the base and base-TC configurations based on MWUT. The column "mean LCF" shows the mean proportion of LCF nodes among non-constant leaf nodes. The column "mean consts" shows the mean proportion of constant leaf nodes among all leaf nodes.

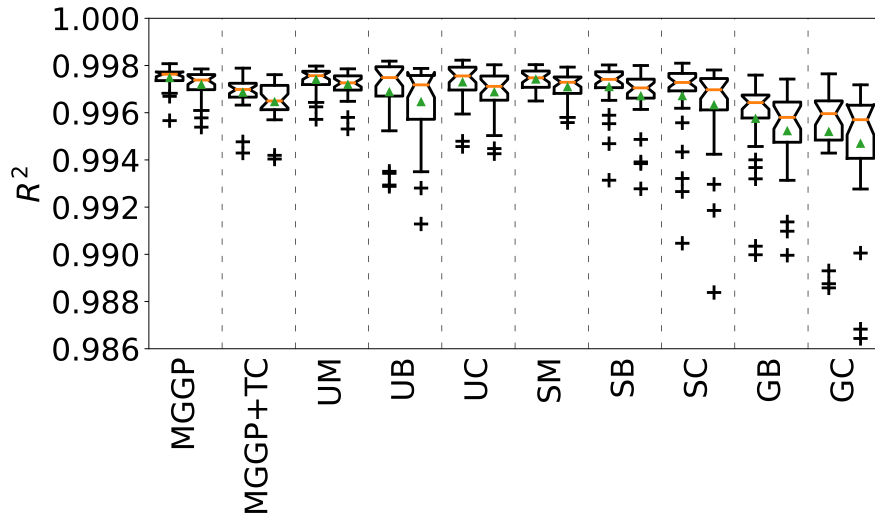


Figure 5.10. Training (left) and testing (right) R^2 boxplots over all 30 runs on the ENH dataset.

config.	training R^2 median	training R^2 max / min	testing R^2 median	testing R^2 max / min	vs. MGGP	vs. MGGP+TC	mean LCF	mean consts	mean depth
MGGP	0.955	0.988 / 0.879	0.909	0.978 / -0.664			–	0.088	11.00
MGGP+TC	0.913	0.969 / 0.808	0.859	0.960 / 0.719			–	0.156	10.87
UM	0.960	0.993 / 0.877	0.918	0.987 / 0.683			0.527	0.050	11.00
UB	0.985	0.994 / 0.963	0.971	0.994 / 0.881	✓	✓	0.894	0.025	9.73
UC	0.985	0.996 / 0.930	0.966	0.992 / 0.916	✓	✓	0.885	0.026	9.43
SM	0.946	0.987 / 0.846	0.907	0.981 / -0.185			0.528	0.052	11.00
SB	0.977	0.991 / 0.881	0.955	0.984 / 0.819	✓	✓	0.598	0.037	10.37
SC	0.968	0.993 / 0.885	0.958	0.978 / 0.694		✓	0.633	0.050	9.60
GB	0.927	0.983 / 0.727	0.885	0.985 / -5.11e3			0.466	0.039	10.47
GC	0.880	0.961 / 0.805	0.822	0.941 / 0.627	×		0.165	0.044	9.70

Table 5.12. Results on the PS problem. Columns “vs. MGGP” and “vs. MGGP+TC” mark whether the configuration was significantly better, worse, or indifferent to the base and base-TC configurations based on MWUT. The column “mean LCF” shows the mean proportion of LCF nodes among non-constant leaf nodes. The column “mean consts” shows the mean proportion of constant leaf nodes among all leaf nodes.

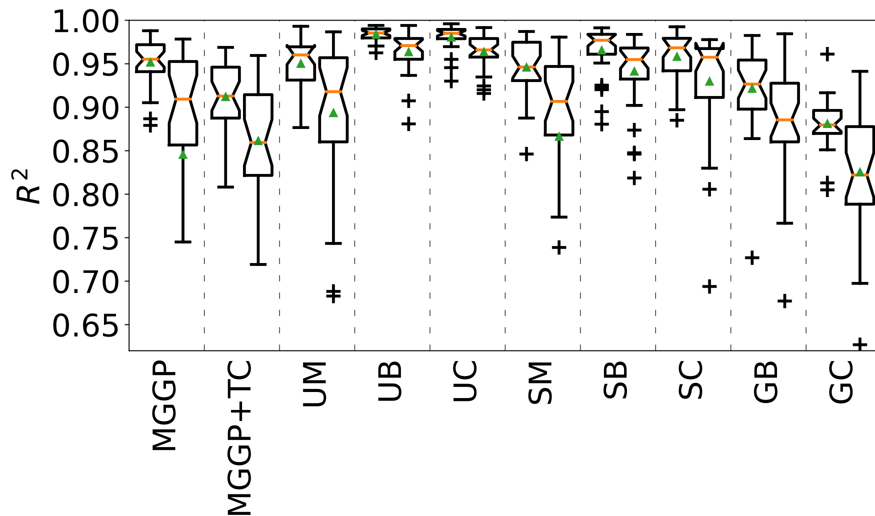


Figure 5.11. Training (left) and testing (right) R^2 boxplots over all 30 runs on the PS dataset. Note that multiple outliers are not shown: at -0.664 for base, at -0.185 for SM, and at -5.11e3 for GB, all of them for test set.

config.	training R^2		testing R^2		vs. MGGP	vs. MGGP+TC	mean LCF	mean consts	mean depth
	median	max min	median	max min					
MGGP	0.931	0.979 0.841	0.885	0.970 0.175			–	0.086	11.00
MGGP+TC	0.908	0.969 0.728	0.864	0.959 -4.771			–	0.171	10.93
UM	0.938	0.992 0.886	0.899	0.975 0.606		✓	0.517	0.046	10.97
UB	0.970	0.993 0.938	0.955	0.987 0.886	✓	✓	0.895	0.030	8.93
UC	0.976	0.991 0.915	0.962	0.988 0.865	✓	✓	0.912	0.019	8.77
SM	0.937	0.989 0.851	0.910	0.980 -6.844			0.498	0.069	11.00
SB	0.942	0.988 0.884	0.928	0.992 0.769		✓	0.569	0.028	10.37
SC	0.952	0.989 0.836	0.931	0.990 0.788	✓	✓	0.623	0.029	10.00
GB	0.887	0.966 0.694	0.853	0.967 0.505			0.518	0.051	10.27
GC	0.862	0.925 0.768	0.829	0.915 -2.86e3	×		0.151	0.041	10.10

Table 5.13. Results on the PS-I problem. Columns "vs. MGGP" and "vs. MGGP+TC" mark whether the configuration was significantly better, worse, or indifferent to the base and base-TC configurations based on MWUT. The column "mean LCF" shows the mean proportion of LCF nodes among non-constant leaf nodes. The column "mean consts" shows the mean proportion of constant leaf nodes among all leaf nodes.

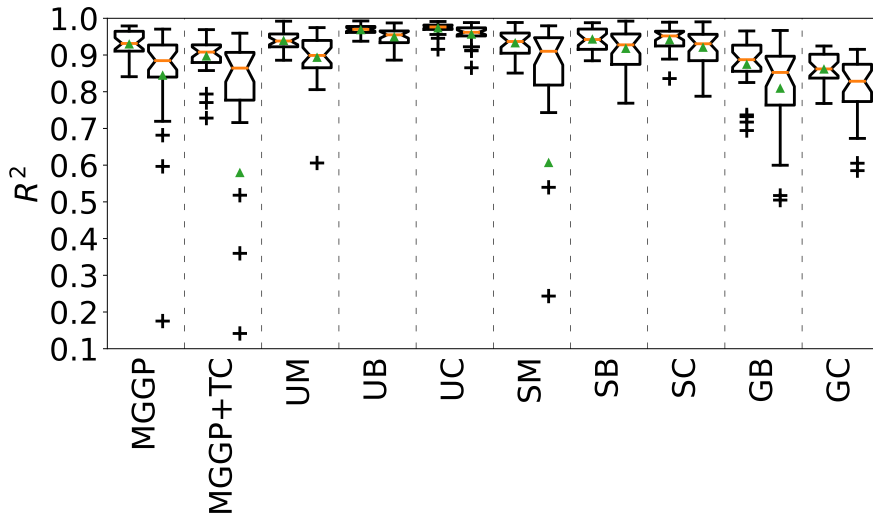


Figure 5.12. Training (left) and testing (right) R^2 boxplots over all 30 runs on the PS-I dataset. Note that multiple outliers are not shown: at -4.771 and -0.580 for base-TC, at -6.844 for SM, and at -2.86e3 for GC, all of them for test set.

config.	training R^2		testing R^2		vs. MGGP	vs. MGGP+TC	mean LCF	mean consts	mean depth
	median	max min	median	max min					
MGGP	0.966	0.987 0.954	0.960	0.983 0.930			–	0.151	11.00
MGGP+TC	0.966	0.987 0.948	0.958	0.982 0.939			–	0.231	10.93
UM	0.970	0.988 0.957	0.961	0.987 0.947			0.552	0.085	11.00
UB	0.988	0.997 0.973	0.985	0.995 0.969	✓	✓	0.763	0.032	9.43
UC	0.988	0.996 0.969	0.985	0.995 0.943	✓	✓	0.797	0.038	9.03
SM	0.970	0.989 0.958	0.965	0.982 0.937			0.565	0.098	11.00
SB	0.976	0.991 0.967	0.973	0.986 0.961	✓	✓	0.559	0.040	9.77
SC	0.974	0.997 0.947	0.971	0.996 0.935	✓	✓	0.563	0.067	10.30
GB	0.967	0.982 0.941	0.959	0.981 0.928			0.462	0.076	10.40
GC	0.961	0.977 0.942	0.952	0.969 0.928		×	0.248	0.071	10.03

Table 5.14. Results on the MM problem. Columns “vs. MGGP” and “vs. MGGP+TC” mark whether the configuration was significantly better, worse, or indifferent to the base and base-TC configurations based on MWUT. The column “mean LCF” shows the mean proportion of LCF nodes among non-constant leaf nodes. The column “mean consts” shows the mean proportion of constant leaf nodes among all leaf nodes.

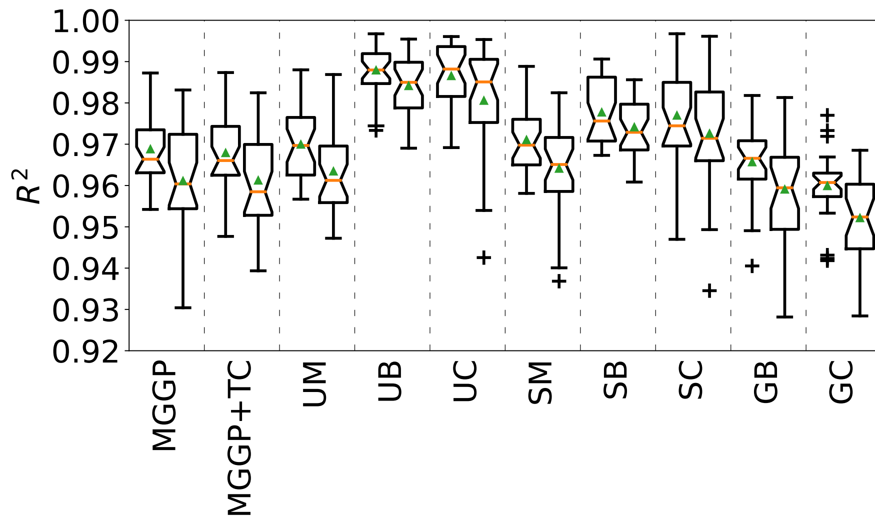


Figure 5.13. Training (left) and testing (right) R^2 boxplots over all 30 runs on the MM dataset.

5.6 Answers to research questions

RQ1: Are LCFs beneficial when an affine transformation is truly present in the data? Yes, LCFs are beneficial in this case as the results on the toy problem have revealed. Five out of eight configurations with LCFs were shown to be significantly better than an unmodified MGGP. Neither of the configurations that used only mutation to tune the LCF parameters were among the successful ones, and the configuration with globally synchronised mode and combined gradient-based and random mutation tuning was even significantly worse.

RQ2: Are LCFs beneficial in other cases? There is no clear answer to this question. LCF-enabled configurations were successful on the K11C, CCS, PS, PS-I, and MM datasets and indifferent to or worse than the baseline on the UB5D, ASN, ENC, and ENH datasets. Of the datasets where a success was recorded, K11C is an artificial dataset where the defining function has elements that are suitable to LCFs, and PS, PS-I, and MM datasets are all datasets from reinforcement learning domain of which PS and PS-I clearly show features (especially the central diagonal "wall" in PS, see Figure 5.3) that could be modelled easier with an affine transformation available.

RQ3: Are LCFs with gradient-based tuning better than using the same kind of tuning on numeric constants? Yes – in no experiment was the MGGP+TC configuration (the configuration with numeric leaf nodes tuned by gradient-based tuning) significantly better than the MGGP baseline, while configurations with LCFs have shown improvement, especially the UB and UC configurations which both employ gradient-based tuning. Since LCF-enabled configurations with mutation-based tuning only were not as successful, it can be argued that it is the combination of LCFs and gradient-based tuning that makes the difference.

RQ4: Which algorithm configuration with LCFs should be adopted as the best one, or the default, if any? There were two configurations that were similarly successful. Both of them use the unsynchronized mode and gradient-based tuning and they differ only in that one uses also mutation of the LCF parameters while the other one does not. We therefore conclude that these configurations should be the first choice if one decides to use LCFs.

5.7 Summary and conclusions

In this chapter we have presented a new type of leaf node for use in SR – a linear combination of feature variables, or LCF. The coefficients of these linear combinations can be tuned in two major ways – by random mutation, or by a gradient-based approach which utilizes a form of backpropagation algorithm and an update rule to adjust the weights accordingly. We also presented three operation modes: an "unsynchronized" mode where there are no restrictions put on the coefficients of the LCF nodes, a "synchronized" mode where all LCFs in an individual form a single affine transformation, and a "globally synchronized" mode where all LCFs across the whole population form a single such transformation.

We have performed two sets of experiments. In the first set we used a very simple problem to test the assumption that LCFs can help when an affine transformation is actually present in the data. The results on this toy problem confirmed this assumption as the algorithm configurations with tuned LCFs (with the single exception of the GC

configuration) were able to produce (almost) perfectly fitting models while the other ones were not.

In the second set of experiments, we have run the algorithm configurations on multiple datasets, both artificial and real-world. The results have shown that both "unsynchronized" configurations with gradient-based tuning (UB and UC) were the best performers as they improved upon MGGP on 5 out of 9 datasets, and were detrimental only on a single dataset. We have shown that gradient-based tuning itself is not the cause of the improvement as a control configuration where numerical constants were tuned this way did not improve upon MGGP but, in fact, showed worse performance.

We consider the result a significant one and the LCFs a worthy extension of MGGP and possibly other GP-based algorithms.

Chapter 6
Feature space transformations in dynamic scenarios

In Chapter 5 we have examined the benefit of using LCFs in a classical static regression scenario. In this Chapter we examine another application of this technique, that is dynamic scenarios where the target is gradually changing.

First, we cover the basics of dynamic optimization and related research in this field. Next, we review existing research regarding dynamic SR scenarios. We then proceed with introducing the problem we focus on in this chapter – dynamic SR scenarios with gradually changing targets – for which we propose to use LCFs. Finally, we perform extensive experimental evaluation of this idea and discuss the results and implications.

6.1 Evolutionary Dynamic optimization

Dynamic optimisation refers to an optimisation task where some characteristics of the solved problem are not static and change during the optimisation process. One such case is when the optimum of the optimisation task is changing, i.e. when the optimisation algorithm is *chasing a moving target*.

There are many flavours of dynamic optimisation tasks which differ in the dynamics of the changes. If the target changes only once in a very long time frame (compared to the time required to find the solution), the dynamic task can be considered just a sequence of static optimisation tasks. On the other side of the spectrum, there is a case where the target is changing all the time (e.g. every iteration of the algorithm, or even between individual evaluations). Here we aim in between these two extremes: the target changes at discrete time steps, thus forming a sequence of separate stages, slowly enough that there is a reasonable amount of time for the algorithm to work on each stage, but fast enough that it cannot be considered a sequence of static tasks.

In evolutionary dynamic optimization (EDO) [64], there are two dominant classes of approaches that deal with changing environments: using a memory and maintaining diversity. Memory-based approaches are suitable for problems with recurring behaviour. The core idea is that information discovered previously might be useful in the future. The memory can be implicit by utilizing a redundant representation of the individuals (e.g. [65–66]), or explicit where a specific information is stored in a dedicated memory and later reintroduced into the population [67–70]. Since we are not dealing with problems with recurring behaviour, we omit memory-based approaches from further consideration.

The approaches based on maintaining diversity aim to counteract the convergence of the population to a small area of the search space, thus allowing the algorithm to refocus to a different area in the search space when the environment changes. Such approaches are more suitable for our setting with gradual changes. Hypermutation [71] is an adaptive mutation operator that increases the mutation rate for a period of time after a change is detected. Variable local search (VLS) [72–73] also reacts on a detected

change, but instead of increasing the mutation rate it increases the mutation strength or size. The Random Immigrants technique [74] introduces several randomly generated individuals to the population in each generation, thus keeping the diversity. Fitness sharing [75–76] maintains the diversity by penalizing similar individuals. Diversity can be also maintained by making it an objective and optimizing both for fitness and diversity using a multi-objective algorithm [77].

6.2 Dynamic symbolic regression

SR in dynamic scenarios has not been studied very intensively so far, we found only a few occurrences in the literature.

In [78], with the aim to propose a dynamic benchmark for GP, the authors examined a two-stage SR scenario where the target function differed in the two stages of the run. They tested whether it is better to restart the algorithm with the new stage, or continue with the current population, in relation to the distance between the two targets. The results suggest that the more distant the two targets are, the more beneficial it is to restart the algorithm.

In [79] the focus is specifically on GP in dynamic environments and techniques improving its behaviour in such scenarios. One of the testing problems was SR with the target defined as $\sum_{i=1}^{8} a_i x^i$ with the dynamic behaviour being realized by switching (some of) the coefficients a_i from +1 to -1 and back over time in three difficulty settings. In the "easy" setting the target alternates at regular intervals between two such polynomials differing in just one coefficient. In the "medium" setting the target cycles probabilistically between five such polynomials. In the "hard" setting the target changes at random times to a random such polynomial (each coefficient switched with 50% probability). The authors put several base techniques to the test (hypermutation [71], an adaptation of an immigrants-based technique ERIGA [80], a memory approach where the best individual is stored and reintroduced into the population when the environment changes, and an adaptation of a technique called Transformation [81]) as well as hybrid techniques that combine two or more of the base techniques. According to the results to this particular problem, the best technique was a hybrid of hypermutation, memory, transformation and standard GP combined via multiple sub-populations.

In [82] the authors focus on artificially introducing dynamics to a static SR problem as a means to enhance the fitting process. The idea is to start with only a few samples from the training set, and add the rest gradually over the course of the run, up to the point when the full set is used. This approach is reported to produce higher quality solutions of smaller size than when using the full training set from the start. The authors also propose an immigrants-based technique using a Kendal Tau distance measure to assess the severity of the environment change, which is then used to determine the number of immigrants introduced into the population.

In [83] the author examines the impact of the size of the population and using the ALPS technique in Cartesian GP (CGP) in dynamic SR scenarios. The scenarios are constructed as switching the target between several 1-dimensional polynomials and their ratios. The results suggest that using ALPS is superior to ordinary CGP regardless of the population size. Also, the author concludes that if an abrupt change takes place, it is better to restart the algorithm from scratch.

Note that all the above-mentioned instances of dynamic SR tasks found in the literature use very dissimilar targets between stages, with the exception of [82] where the

target function does not change at all (only the training dataset changes). These results are thus not directly relevant for our class of tasks with smaller, gradual changes.

6.3 Dynamic scenarios with gradual changes

Basically all the research in dynamic SR mentioned in the previous section has only a very little connection to real-world use cases. All the benchmarks are artificial, often constructed by switching between more or less arbitrarily chosen functions. In this work we use dynamic SR problems with a close relation to a real-world use case – Reinforcement Learning (RL). Our benchmarks (which are discussed in detail in Section 6.6.2) simulate the search for a value function of four RL benchmark problems, and we strongly believe that their similarity to the real-world dynamic SR tasks with gradual changes is much higher than the similarity of the problems found in the literature.

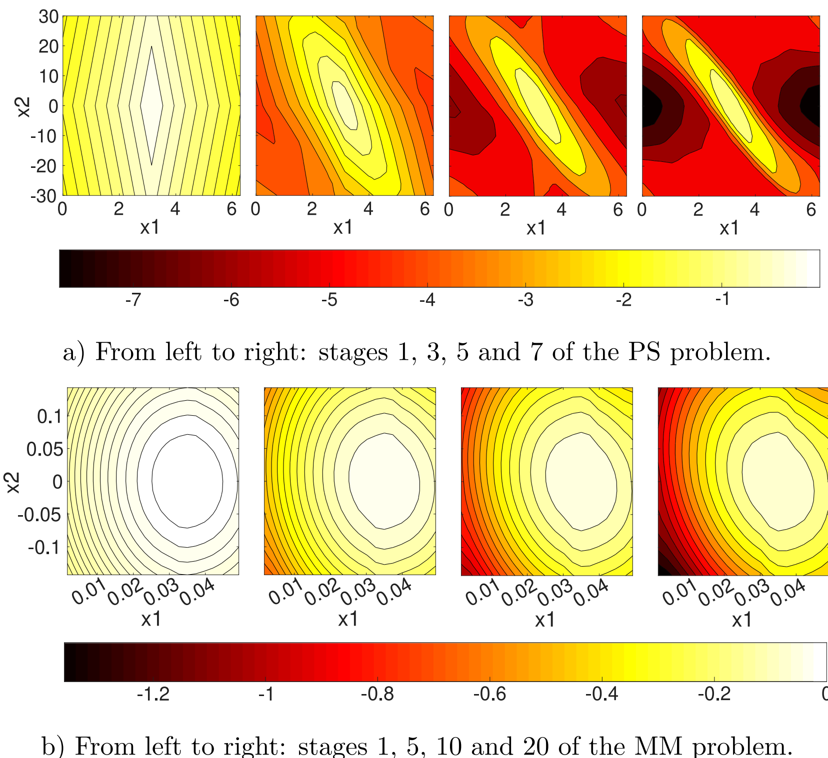
A *value function* (V-function for short) is a function that maps a state from the state space of the RL problem to an expected return given a policy. Many RL solution methods involve finding the *optimal* V-function (which provides state values given the *optimal* policy). One method for finding the optimal V-function is the Value Iteration algorithm (VIA). VIA works by iteratively improving the V-function starting from a trivial estimate (usually just the reward function or zero) and updating the value for each state by maximizing the value from the neighbouring states. When the state space is finite and small enough, the V-function can simply be represented as a lookup table with the value for each state. However, for continuous state spaces, unless discretization is used, the V-function must truly be a function rather than a table. Conventional methods employ some kind of numeric approximator to represent the function, e.g. simple linear or polynomial models, neural networks, or basis functions [84]. The question whether these approximators can be replaced by a symbolic model learned by SR is an interesting research topic (beyond the scope of this work), and the origin of our benchmark problems. The benchmarks thus require the dynamic SR algorithm to find a symbolic approximator of a sequence of targets precomputed by VIA, as a check how well SR can fit the data coming from the VIA.

In this work we aim at gradually changing target functions. The sought function may be, for example, rotated or scaled in one or more directions, but still be close to the previous one. The general shape should be more or less kept, without major qualitative changes. The benchmarks we focus on shall have the following properties:

- a benchmark has a number of stages (each represented by a different dataset);
- the features of all data points are fixed across all stages, i.e. the points do not "move" across the feature space[1];
- the stages differ only in the target function values of the data points;
- the stages (the target values of all data points) are switched in regular time intervals, regardless of the actual performance of the algorithm, and the algorithm's job is to adapt to these new targets;
- the changes from stage to stage are gradual.

An example of such data can be seen in Figure 6.1 which displays four selected stages from two of the benchmarks used later in this chapter (described in detail in Section 6.6.2).

[1] The algorithms that we subject to testing later in this chapter (see Sections 6.4 and 6.6) are, in fact, indifferent to how the training data changes. In principle, the whole dataset could be replaced by a completely different one. However, that would be a different problem, while we want to focus on the described kind of dynamics.



a) From left to right: stages 1, 3, 5 and 7 of the PS problem.

b) From left to right: stages 1, 5, 10 and 20 of the MM problem.

Figure 6.1. Example stages of two of the benchmarks. Note that the contour plot is used to better show the shape but in fact the benchmarks are defined by a set of data points, not by an actual function which is unknown.

In the first row of the figure, we can clearly see a trend where the starting shape, a very simple one, is smoothed out and rotated. In the second row, the changes are even smaller. We can see that the initial shape is generally kept, but it is then deformed as if "sheared" by pulling the right-hand side of the shape towards the negative part of axis x_2 and the left-hand side is pulled in the opposite direction.

To further illustrate the size of the differences between individual stages of our benchmarks, Figure 6.2 shows how each stage differs from the previous one for all benchmarks. The metric used to measure this difference is $1 - |corr_{ss}|$ where $corr_{ss}$ is the Pearson correlation of two consecutive stages. In [78] a mean squared distance was used as a difference measure; nevertheless, we have chosen the correlation-based measure since it "hides" simple linear scaling and shift of the target values which is desirable in case of scaled GP, or MGGP (which we use as the base algorithm), because the top-level linear regression takes care of such differences easily. The correlation value is subtracted from 1 for better visualization (and, thus, represents a dissimilarity of two targets, or stages).

Beside the used benchmarks (artificial, PS, PS2, MM, MM2), the dissimilarity distributions for two extra datasets are displayed in Figure 6.2 as a reference in the form of the median, first quartile and third quartile of the correlation value they exhibit. The first one is a random dataset – a dataset whose targets are uniformly randomly resampled every stage. It is no surprise that such dataset exhibits almost zero correlation between stages. The second one is the 8th degree polynomial regression problem from [79], already mentioned in Section 6.2. The displayed values are aggregation of correlation values of all possible transitions between the 256 possible states used in that problem (i.e. switching +/- signs in front of the polynomial's elements).

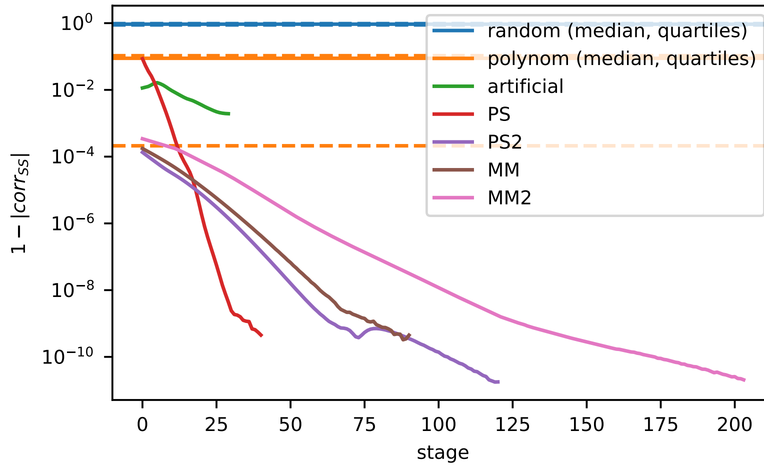


Figure 6.2. Dissimilarity of targets of neighbouring stages for each of the benchmarks. The higher the value, the less similar the neighbouring stages are.

We can see that for our benchmarks the differences between consecutive stages get smaller with increasing stage number which is an expected phenomenon for datasets generated by VIA because the values should converge towards the true value function. It is also important to note that the correlation-based dissimilarity measures only how close the targets are, but not necessarily how easy it is for an algorithm to track these targets, or how useful the LCFs are going to be.

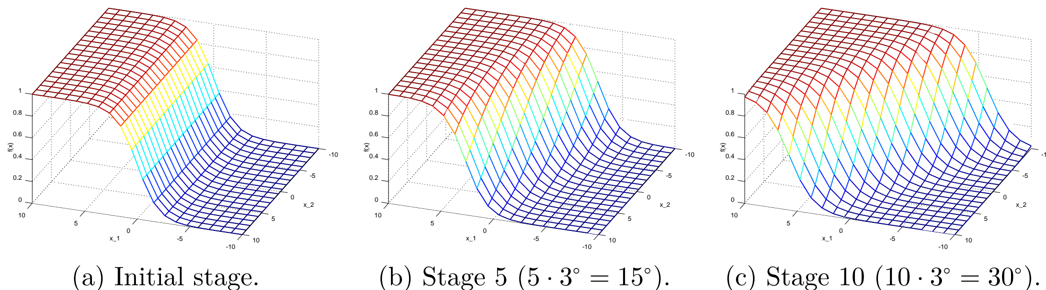
6.4 Proof-of-concept experiment

Before we dive into proper evaluation on serious benchmarks, we demonstrate the LCFs in action with a minimalistic experiment we designed for this purpose. We use an artificial benchmark where the true function is known and therefore can be compared to the evolved models. The benchmark is a 7-dimensional dataset with 31 stages, i.e. 31 different target values for each datapoint that change from stage to stage. The target values of the first stage are given by a sigmoid function along the first dimension, i.e. the value of the function is independent of the other dimensions. In each of the following stages, the feature vector is rotated by $\frac{\pi}{60}$ rad (or 3°) in all pairs of axes, and the first dimension of this rotated vector is passed to the sigmoid function. At the last (31st) stage, the feature space has rotated by $\frac{\pi}{2}$ rad (or 90°) in all pairs of axes and therefore the values now again depend only on one of the features. For illustration, a 2-dimensional example on regular grid is shown in Figure 6.3.

Effectively, the values in stage s are defined by the sigmoid function

$$f_s(\mathbf{x}) = \frac{1}{1 + e^{\mathbf{r_s}^\top \cdot \mathbf{x}}}$$

where $\mathbf{x}$ is the feature vector, and $\mathbf{r_s}$ is a vector from the rotation matrix of the s-th stage corresponding to the first dimension. The data points come from a 7-dimensional space and are sampled uniformly from the interval $[-10, 10]^7$. For the sake of this experiment we select three stages $s = 1$, $s = 2$ and $s = 15$ to examine the models produced by the



(a) Initial stage. (b) Stage 5 ($5 \cdot 3^\circ = 15^\circ$). (c) Stage 10 ($10 \cdot 3^\circ = 30^\circ$).

Figure 6.3. A 2-dimensional example of gradually rotated sigmoid function on a regular grid. Note that in the actual benchmark there are 7 dimensions and the points are sampled randomly.

algorithm. The values of $\mathbf{r_s}$ for these stages are (rounded to three significant figures):

$$\mathbf{r_1} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \qquad \mathbf{r_2} = \begin{bmatrix} 0.992 \\ 0.0520 \\ 0.0520 \\ 0.0521 \\ 0.0522 \\ 0.0523 \\ 0.0523 \end{bmatrix} \qquad \mathbf{r_{15}} = \begin{bmatrix} 0.168 \\ 0.152 \\ 0.204 \\ 0.275 \\ 0.370 \\ 0.497 \\ 0.669 \end{bmatrix}$$

For this proof-of-concept experiment, the underlying algorithm is vanilla GP with ephemeral random constants, i.e. no MGGP and no linear fitting of the output of the tree. The function set is identical to the one used for the actual comparative experiments and is described in Section 6.6.3. Note that it includes the very sigmoid function so it is easy to find the correct model if a suitable argument to the function is found too. This algorithm is run both with LCFs and without them. The goal is to look at the models produced by the algorithms and compare them to the true function.

Stage 1 (initial stage). In the first stage, both the algorithms with and without LCFs produced models equal to the actual true function, as expected:

$$f(\mathbf{x}) = \text{sigmoid}(x_1) \tag{6.1}$$

Stage 2. A typical model produced by GP without LCFs is either the same as in the first stage – the rotation is still small, so the resulting error is also small enough for the first stage model to “survive” – or it is similar to the following function (which is one of the actual models found):

$$f(\mathbf{x}) = \text{sigmoid}(x_1 + \tanh((((\text{sinc}(x_3 + \tanh(\cos(x_4))))^6)^6)^5)) \tag{6.2}$$

where $\text{sinc}(x) = \frac{\sin x}{x}$. We can see it is a sigmoid of the first dimension plus a non-linear function of (some of) the other dimensions. This particular model had $R^2 \approx 0.974$ on testing data. Other models found by GP without LCFs have this inner non-linear function different from the one above but this general form is similar.

GP with LCFs, on the other hand, produced models that almost exactly match the true function:

$$f(\mathbf{x}) = \text{sigmoid}(LCF(\mathbf{x}; \mathbf{a}, 4.772 \cdot 10^{-7})) \tag{6.3}$$

where $\mathbf{a} = \begin{bmatrix} 0.992 & 0.0520 & 0.0521 & 0.0521 & 0.0522 & 0.0523 & 0.0523 \end{bmatrix}^\top$. Compare with $\mathbf{r_2}$ above to see it matches almost exactly. This particular model had $R^2 \approx 1$ on testing data.

Stage 15. A typical model produced by GP without LCFs looks like this:

$$\begin{aligned} f(\mathbf{x}) = \text{sigmoid}(&x_7 \cdot \text{sigmoid}((\tanh(x_7))^2 \cdot x_6^2) \cdot \text{sigmoid}(\tanh(x_7) \cdot x_6 \cdot \\ &\text{sigmoid}(\text{sigmoid}(\text{sigmoid}(x_3))) \cdot \text{sigmoid}(\text{sigmoid}(x_5 \cdot x_6)))) \end{aligned} \tag{6.4}$$

This particular model had $R^2 \approx 0.575$ on testing data.

GP with LCFs, again, produces models almost perfectly matching the ground truth:

$$f(\mathbf{x}) = \text{sigmoid}(LCF(\mathbf{x}; \mathbf{a}, 5.304 \cdot 10^{-8}))) \tag{6.5}$$

where $\mathbf{a} = \begin{bmatrix} 0.168 & 0.152 & 0.204 & 0.275 & 0.370 & 0.497 & 0.669 \end{bmatrix}^\top$. This particular model had $R^2 \approx 1$ on testing data.

We can see that LCFs are indeed able (at least in principle) to find a suitable transformation of the feature space. Although the GP algorithm can create models with multiple LCFs, the best models found in stages 2 and 15 contain only a single LCF (which is sufficient to solve this particular problem).

Note that the kind of analysis presented in this section can be done only in cases where the true function in each stage is known. This is not the case for our other RL benchmarks, where only the datasets for individual stages are available, but the underlying function is unknown. Also note that even though the true function is as simple as the one used in this minimalistic experiment, the produced models may not match as perfectly, if a different underlying algorithm than vanilla GP is used simply because such algorithm can have more ways to express the function, or it is easier for the algorithm to do so (e.g. MGGP models used in this work can have more trees and the found transformation can be "distributed" among them).

6.5 Research questions

With the proof-of-concept experiment showing that LCFs can be, in principle, beneficial, we state four research questions regarding how LCFs plugged into MGGP are going to perform on more realistic benchmarks. The research questions are:

- RQ1: Does MGGP with LCFs enable better target tracking compared to MGGP without them?
- RQ2: Is MGGP with LCFs better than just restarting the ordinary MGGP with each change of the target?
- RQ3: Is it profitable for LCFs to retain information learned in previous stages, or is it better to start learning it from scratch after each target change?
- RQ4: Is the use of LCFs better than a classic evolutionary dynamic optimisation method, hypermutation, applied to an ordinary MGGP?

6.6 Experimental evaluation

6.6.1 Algorithm configuration

We have designed six configurations that, when compared one to another, should exhibit the benefits or drawbacks of the LCFs and answer the research questions. The configurations combine the base algorithm with three different techniques: LCFs as presented

in Section 5.1, restarting as a naïve way of dealing with changes, and hypermutation as a representative of the classic EDO techniques. Hypermutation was chosen because it is a well-established technique, and because (from [64], Section 4.3.2.): "... methods like hyper-mutation appear to be good in solving problems with highly frequent changes where changes are small and medium." The configurations are:

- MGGP
- MGGP + LCF
- restarted MGGP
- restarted MGGP + LCF
- MGGP with hypermutation
- MGGP + LCF with hypermutation

MGGP. It is a pure MGGP implementation without any of our modifications, i.e. the individuals are composed of one or more genes that are combined via the top-level linear regression, and the trees are evolved genetically (i.e. using crossover and structural and gaussian mutations). The algorithm does not explicitly react on the changed target values in any way (it only starts to get different fitness values).

MGGP + LCF. It is MGGP extended by adding LCFs (as introduced in Chapter 5, see Section 5.1) to the pool of terminal nodes available for construction of trees. The coefficients at these nodes are tuned by several steps of backpropagation and iRprop$^-$ update algorithm just before the fitness evaluation. The LCFs are used in the *unsynchronized* mode (see Section 5.1.3) as this was the most successful mode in the static problems in previous chapter (see Section 5.5.7). This algorithm does not explicitly react on the changed target either. Results of this algorithm should, when compared to MGGP, show whether LCFs bring an improvement or not (RQ1).

Restarted MGGP, restarted MGGP + LCF. Identical to MGGP and MGGP + LCF respectively, except that after each change of the target the whole population is discarded (including already optimized LCF nodes) and a new one is initialized from scratch. The algorithm is effectively restarted. Results of restarted MGGP should, when compared to MGGP + LCF, show whether LCFs are actually effective or if a mere restart is enough (RQ2). Results of restarted MGGP + LCF should, when compared to MGGP + LCF, show whether the use of information learned by LCFs in the previous stages is beneficial for the algorithm (RQ3).

MGGP with hypermutation, MGGP + LCF with hypermutation. Identical to MGGP and MGGP + LCF respectively, except that after each change a hypermutation (i.e. an increased probability of mutation) is employed for a fixed amount of time. Results of MGGP with hypermutation should, when compared to MGGP + LCF, show how LCFs compare to this classic EDO technique (RQ4). The MGGP + LCF with hypermutation is included for completeness, and the results should show whether the benefit of both techniques (if any) can be combined to form even better performing setup.

6.6.2 Benchmarks

We use 5 dynamic benchmark problems in our experiments. One of them is the artificial benchmark already described and used in Section 6.4. The other four benchmarks come from the RL domain and they represent different iterations of a Value Iteration algorithm (VIA), taken from a precomputed run of VIA using a conventional numeric approximator. The goal here is *not* to perform VIA with SR used for finding the model

of the V-function, but rather to assess how well our GP improvements cope with the kind of dynamics of the changing target that can be seen in RL problems. For details about the exact process of how these V-functions are obtained see Appendix B.[2] The features of the training data for each of the benchmarks directly correspond to the state space of the corresponding RL problem, i.e. they are samples of states from this space. The samples are placed in a regular grid over the state space. We have chosen four RL tasks:

- Pendulum swingup (PS) – An inverted pendulum is to be swung up to an upright vertical position and stop there, while the torque of the motor is insufficient to do so directly. It has to swing to one side and then use the momentum when the pendulum swings back. The state variables are the angle and angular velocity. The V-function (the target for the regression task) of four example stages can be seen in the Figure 6.1a.
- 2-pendulum swingup (PS2) – Similar to the previous problem but at the end of the pendulum there is another pendulum with a motor. The task is to get both pendulums to an upright vertical position and stop there. The state variables are the two angles and angular velocities.
- 2-coil magnetic manipulation (MM) – There are two electromagnetic coils under a rail where a metal ball is rolling. The goal is to use the magnetic force of the two coils to drive the ball to a specific position. The state variables are the position and velocity of the ball. Four example stages can be seen in the Figure 6.1b.
- 2-coil magnetic manipulation with current (MM2) – Identical to the previous task but there are two additional state variables representing the actual electric currents flowing through the coils.

The main features of all the benchmarks are summarized in Table 6.1. The training and testing data for the artificial benchmark are just different random samplings of the feature space. The training and testing data for the VIA benchmarks differ in the resolution of the grid with the testing data using a finer resolution in each of the dimensions of the state space, therefore (some of) the samples from the training data are also present in the testing data.[3]

task	# of dimensions	# of stages	# of samples		
			training	testing	common
artificial	7	16	700	1750	0
PS	2	42	961	3721	961
PS2	4	122	14364	234526	1936
MM	2	92	441	1681	441
MM2	4	205	3825	61200	400

Table 6.1. Summary of RL benchmarks. The number of dimensions is the number of state variables of the problem. The last column shows how many samples are common for both the training and testing set.

[2] The actual data for all stages of all used problems are available at `https://github.com/zegkljan/dynamic-sr-supplementary`.

[3] For PS and MM benchmarks the spacing between the grid points in the testing set is half of the spacing in the training set and therefore all the points in the training set is also present in the testing set. For PS2 and MM2 benchmarks the number of points in each dimension in the testing set is double the number of points in the corresponding dimensions in the training set and therefore some but not all of the training set points are also present in the testing set.

6.6.3 Experiment settings

The fitness measure used by all the algorithm configurations is the coefficient of determination R^2. In each run of the algorithm on a benchmark, the individual stages of the benchmarks are used as targets, sequentially. The targets are switched after a fixed amount of time. For each benchmark and algorithm, 20 runs were performed (with different random seeds). The time per stage was set to 60 seconds for all algorithms and benchmarks except for PS2 and MM2 benchmarks where the time was set to 240 seconds (because of higher number of dimensions as well as the number of samples)[4]. For the artificial benchmark, a different sample from the same distribution was generated for each of the runs while for the VIA benchmarks (PS, PS2, MM, MM2) only one sampling per benchmark was used in all the runs.

Testing environment. We use our own implementation of the algorithms [60] which is written in Python 3 and uses the NumPy library for vector and matrix calculations including the linear regression. All the experiments were carried out on machines with identical configuration[5].

Algorithm parameters. The parameters of the algorithm for all the benchmarks and for all its versions were the same and are summarized in Table 6.2. The parameters were determined during preliminary testing. The set of functions available to the algorithm is the following (using x and y as arguments): $x + y$, $x - y$, $x \cdot y$, $\sin(x)$, $\cos(x)$, e^{x}, $\mathrm{sigmoid}(x) = \frac{1}{1+\mathrm{e}^{\mathrm{x}}}$, $\tanh(x)$, $\mathrm{sinc}(x) = \frac{\sin(x)}{x}$, $\mathrm{softplus}(x) = \ln(1 + \mathrm{e}^{\mathrm{x}})$, $\mathrm{gauss}(x) = \mathrm{e}^{-\mathrm{x}^2}$, x^2, x^3, x^4, x^5, x^6.

6.6.4 Results

The progress of the algorithms across the stages can be seen in Figures 6.4 to 6.8. Here we are primarily interested in the performance at the end of each stage. The plots thus show the performances just before the target is switched to the next stage. (The behaviour within the stages is discussed shortly in Section 6.6.6.) For better context, the plots also show how the targets of the benchmark itself differ between the stages using the dissimilarity measure discussed in Section 6.3.

We also provide a statistical evaluation of the performances of the configurations. The score used is the *best-error-before-change* [64] (originally called *Accuracy* in [85]). The score is defined as

$$E_B = \frac{1}{m} \sum_{i=1}^{m} e_B(i) \tag{6.6}$$

where m is the number of changes (which we call stages in this work) and $e_B(i)$ is the error of the best individual found in the i-th stage before the switch to the next stage. We use the R^2 score in place of e_B and therefore the E_B score is actually a goodness-of-fit metric rather than error metric, and higher values mean better performance. The 20 runs of one configuration on one problem produce 20 E_B values. These are then statistically compared using MWUT. We compared MGGP + LCF with all the other configurations separately for each benchmark problem. The results of this statistical comparison can be seen in Table 6.3.

[4] We use wall-clock time instead of, for example, the number of generations simply because a generation represents a different amount of computation for algorithm with LCFs and without them.

[5] 2x 8-core Intel Xeon E5-2650 v2 2.6 GHz, Debian 8.7 (Jessie), SPECfp2006 [61] score 490 (30.6 per core). However, the algorithms are single-threaded so only one core was utilized.

parameter	value
fitness measure	R^2
pop. size	125
elitism	10 %
max. gene depth	10
max. # of genes	10
tournament size	4 %
crossover prob.	0.7 (∼0.636)
mutation prob.	0.2 (∼0.363)
subtree crossover prob.	0.85
high-level crossover prob.	0.15
high-level crossover rate	0.5
subtree mutation prob.	0.7
gaussian mutation prob.	0.3
gaussian mutation σ	10.0
following parameters apply only for MGGP + LCF (and variants)	
backprop. & update steps	$\max\{20 - \#nodes, 1\}$
η^+	1.2
η^-	0.5
Δ_0	0.1
Δ_{min}	10^{-6}
Δ_{max}	50

Table 6.2. Parameter values for the algorithms. The percentages relate to the population size. The parenthesized values are active during hypermutation, and were derived by multiplying the regular mutation probability by 2 and then both crossover and mutation probability were scaled to sum up to 1. This factor of 2 was determined in preliminary experiments. Parameters η^+, η^-, Δ_0, Δ_{min}, and Δ_{max} are parameters of iRprop$^-$.

6.6.5 Discussion

In the artificial benchmark (Figure 6.4) we clearly see the superiority of the configurations with LCFs over the ones without them. The rotation of the feature space has almost no impact on the configurations with LCFs, on the contrary to the other configurations. It is also interesting that the behaviour of configurations with LCFs does not differ very much. We argue that this is caused by the fact that if there are LCFs available, the sought function is easy to find even from scratch (i.e. without the help of having the solution to the previous stage available), because LCFs enable to easily construct the rotation transformation which is the only part of the artificial benchmark problem that changes between the stages (see also Section 6.6.6). We have already tested this phenomenon in a static scenario with similar artificial problem in previous chapter (see Section 5.5.5) so we are quite confident this is the correct explanation. Restarted MGGP gets worse as the target rotates but gets back to almost perfect fit at the final stages because the targets are then rotated by 90° and therefore aligned with the axes, making it an easy task again.

In the VIA benchmarks, except for PS2, we can see that MGGP + LCF and MGGP + LCF with hypermutation are, overall, the best performers. Also, restarted MGGP is clearly the worst performer. Restarted MGGP + LCF and pure MGGP (non-restarted) are comparable across all VIA benchmarks. The PS2 benchmark

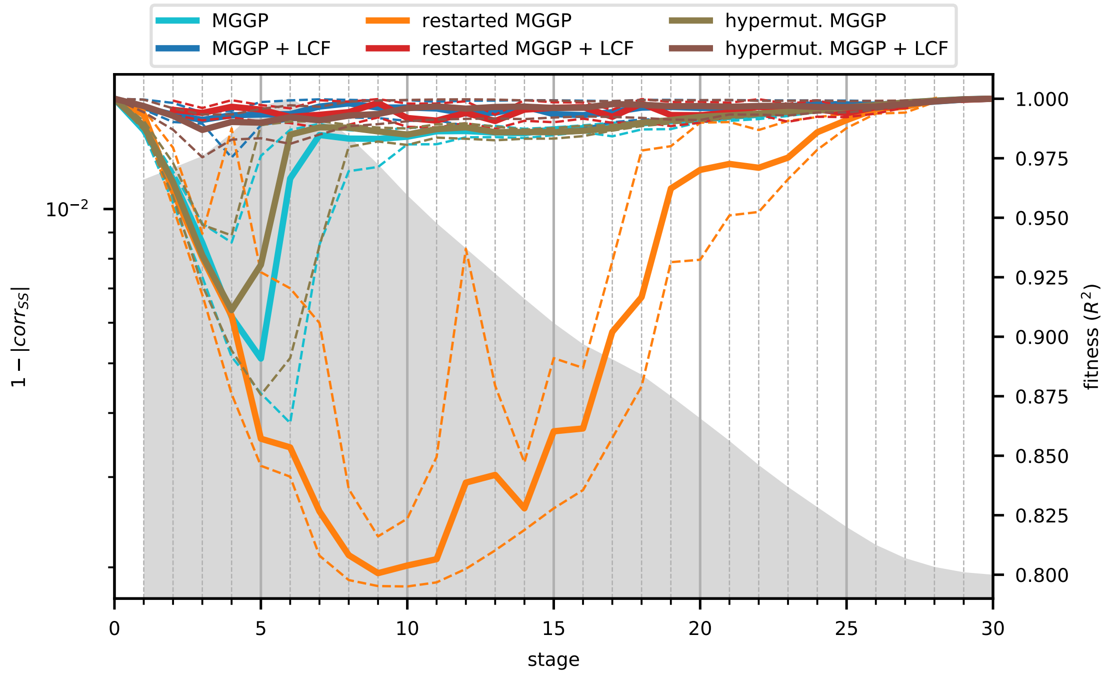


Figure 6.4. Plot of the last-in-stage (i.e. just before the target switch) fitness values over the stages aggregated over 20 runs on the artificial benchmark. Solid lines show the median R^2, dashed lines show the 1st and 3rd quartile of R^2, both on testing data (right axis). Grey area shows the dissimilarity measure (as discussed in Section 6.3) between consecutive stages of the benchmark (left axis).

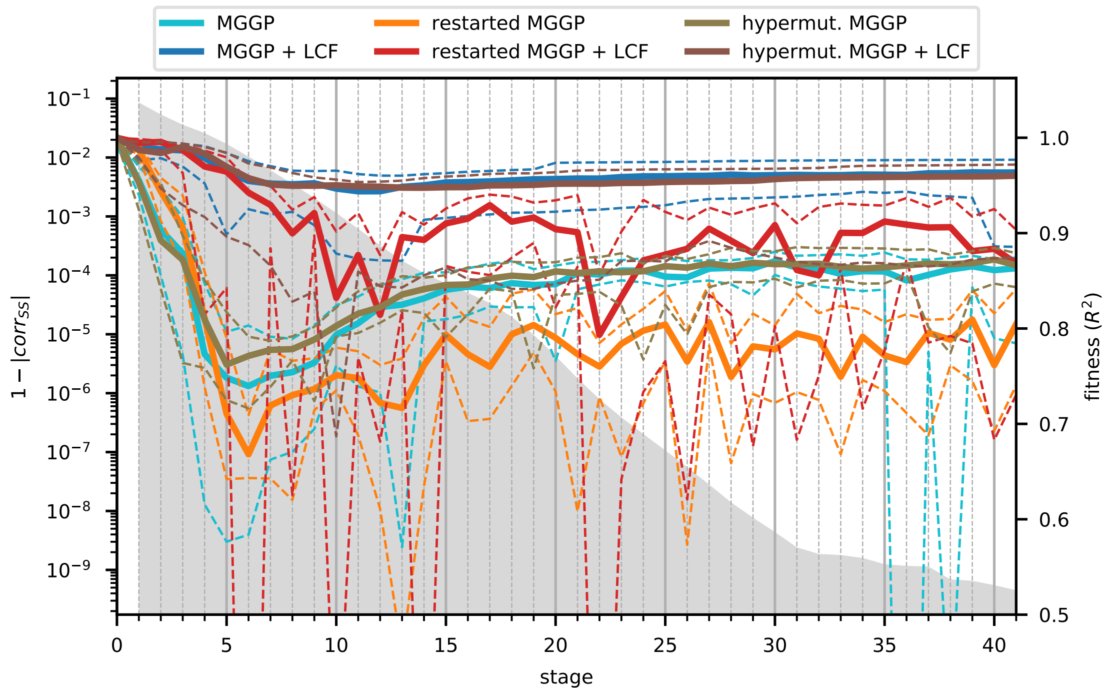


Figure 6.5. Plot of the last-in-stage (i.e. just before the target switch) fitness values over the stages aggregated over 20 runs on the PS benchmark. Solid lines show the median R^2, dashed lines show the 1st and 3rd quartile of R^2, both on testing data (right axis). Grey area shows the dissimilarity measure (as discussed in Section 6.3) between consecutive stages of the benchmark (left axis).

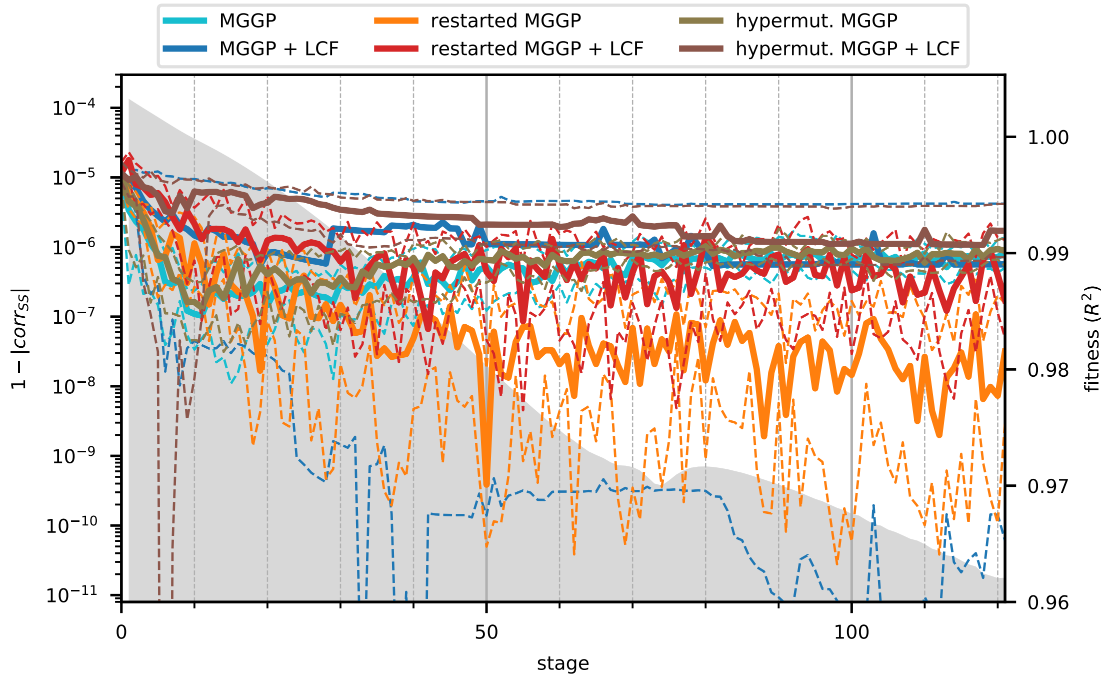


Figure 6.6. Plot of the last-in-stage (i.e. just before the target switch) fitness values over the stages aggregated over 20 runs on the PS2 benchmark. Solid lines show the median R^2, dashed lines show the 1st and 3rd quartile of R^2, both on testing data (right axis). Grey area shows the dissimilarity measure (as discussed in Section 6.3) between consecutive stages of the benchmark (left axis).

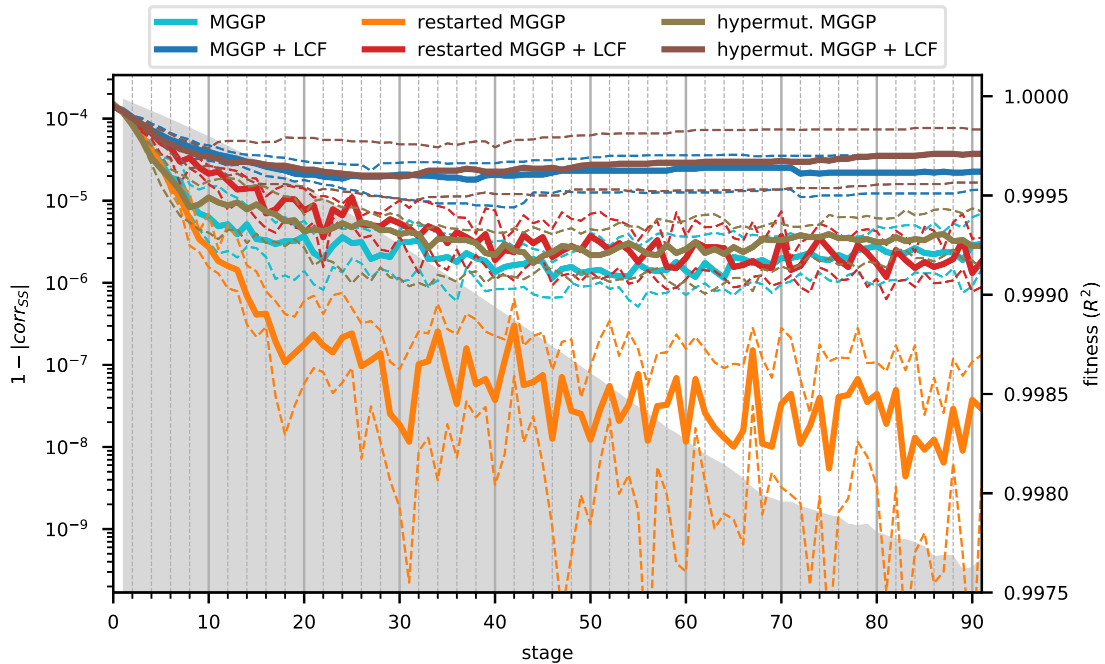


Figure 6.7. Plot of the last-in-stage (i.e. just before the target switch) fitness values over the stages aggregated over 20 runs on the MM benchmark. Solid lines show the median R^2, dashed lines show the 1st and 3rd quartile of R^2, both on testing data (right axis). Grey area shows the dissimilarity measure (as discussed in Section 6.3) between consecutive stages of the benchmark (left axis).

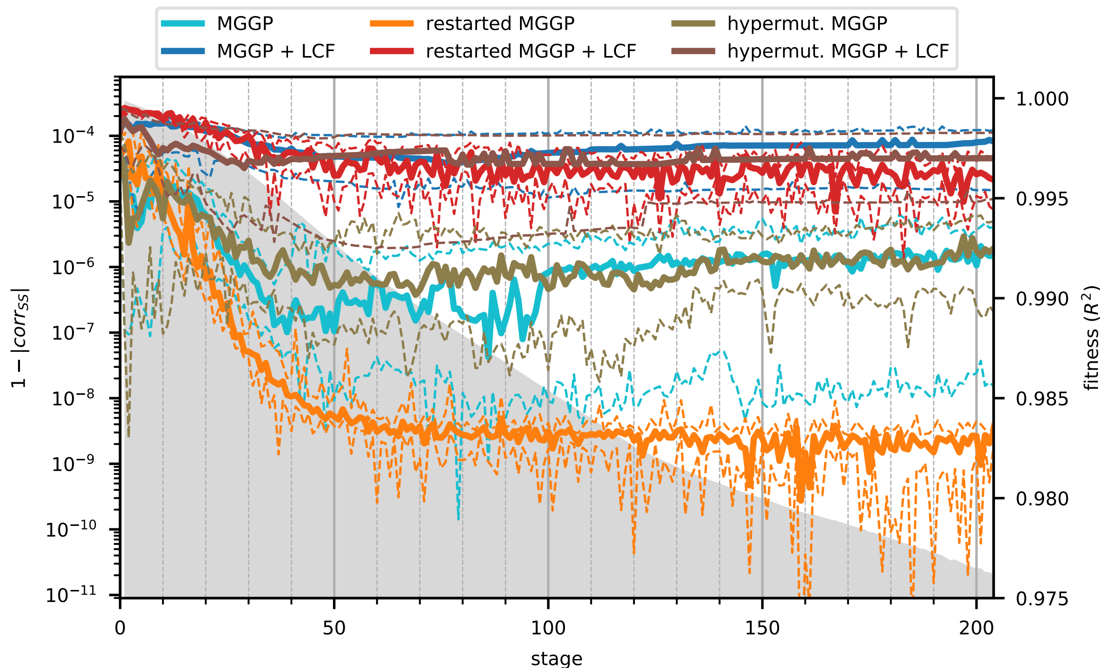


Figure 6.8. Plot of the last-in-stage (i.e. just before the target switch) fitness values over the stages aggregated over 20 runs on the MM2 benchmark. Solid lines show the median R^2, dashed lines show the 1st and 3rd quartile of R^2, both on testing data (right axis). Grey area shows the dissimilarity measure (as discussed in Section 6.3) between consecutive stages of the benchmark (left axis).

turned out to be the only one where MGGP + LCF was not significantly better than any other configuration (see Table 6.3). Yet it still reached better results than other configurations.

A pattern noticeable across all VIA benchmarks is that the variance of the fitness of restarted algorithms across the stages is much higher compared to their non-restarted counterparts. This is probably caused by the fact that during the reinitialization all progress is lost and the algorithm starts from a new random population, easily converging to a different solution. The non-restarted versions, on the other hand, do not lose anything and therefore the transition to the next stage is much smoother.

Note also the worse performance of the restarted algorithms, which shows that the benchmarks are not just sequences of similarly hard static tasks. An important observation visible across all VIA benchmarks (though very minimal for PS2) is that the restarted algorithms get worse with the increasing stage number, recovering only a little. The reason for this phenomenon, we argue, is that the problems actually get more difficult for the restarted versions with the progressing stages. In other words, we argue that finding a model of the targets in a later stage *from scratch* (restarted versions throw away the population after each stage) is more difficult than finding a model of targets in the first stage. As an example, consider the PS benchmark and a depiction of some of its stages we have already presented in Figure 6.1a: in the first stage the shape of the function is symmetrical in both axes around the center point $(\pi, 0)$ and the shape itself is not very complicated. In the initial stages the general shape is still present but is somewhat rotated which requires more manipulation of the features compared to the first stage where the symmetry axes are aligned with the feature space coordinate axes. In the following stages the target becomes more complicated regarding the symmetry, rotation, and the shape, with more pronounced and more localized shape elements. We

configuration	artificial			PS			PS2		
	U	p	r	U	p	r	U	p	r
MGGP	57	**1.16e-4**	5	47	**3.71e-5**	5.5	163	0.323	3
MGGP + LCF	-	-	3	-	-	1	-	-	1
rest. MGGP	20	**1.20e-6**	6	47	**3.71e-5**	5.5	120	3.15e-2	6
rest. MGGP + LCF	223	0.543	1	75	**7.58e-4**	3	152	0.199	5
hyp. MGGP	65	**2.75e-4**	4	69	**4.16e-4**	4	159	0.273	4
hyp. MGGP + LCF	201	0.989	2	175	0.508	2	196	0.925	2

configuration	MM			MM2			avg. rank
	U	p	r	U	p	r	
MGGP	35	**8.60e-6**	5.5	76	**8.36e-4**	6	5
MGGP + LCF	-	-	2	-	-	1	1.6
rest. MGGP	35	**8.60e-6**	5.5	80	**1.23e-3**	5	5.6
rest. MGGP + LCF	40	**1.60e-5**	4	120	3.15e-2	3	3.2
hyp. MGGP	57	**1.16e-4**	3	84	**1.78e-3**	4	3.8
hyp. MGGP + LCF	246	0.218	1	188	0.756	2	1.8

Table 6.3. Statistical comparison of MGGP + LCF to all other algorithms. The U columns show the value of the U statistic for the MGGP + LCF and therefore the smaller the value the better MGGP + LCF was. Possible minimum is 0 (the algorithm was always worse than MGGP + LCF), possible maximum is $20 \cdot 20 = 400$ (the algorithm was always better than MGGP + LCF). The p columns show the p-values of the tests. The values lower than the significance level $\alpha = 0.01$ are typeset in boldface. The r columns show the raking of the algorithms based on the value of the U statistic (the higher the value, and therefore the more times the algorithm was better than MGGP + LCF, the lower the rank). The final column "avg. rank" is the average rank for the algorithm over all benchmarks, the lower the number the better the algorithm.

argue that the first stages are easier to fit than the later ones. Therefore, the restarted algorithms do not keep up with the non-restarted ones because they have to fit the complex shape from scratch compared to the non-restarted ones which already have a fit on a similar shape from the previous stage. To further illustrate this, Figure 6.9 shows the target dissimilarity of a particular stage compared to the first stage in a similar manner we did earlier for stage-to-stage dissimilarity. Since all the algorithms generally have no trouble fitting the first stage, this figure shows the potential difficulty of the benchmarks for the restarted algorithms.

Coming back to the overall results in Figures 6.4 to 6.8, a final notable pattern is the drop of the fitness value related to the change in the target values. We can see that the biggest drop happens when the target values change the most, which is an expected behaviour. However, each of the algorithms reaches the minimum fitness value at a different stage. For example, in the PS, PS2 and MM2 benchmarks, the pure MGGP (in turquoise colour) quickly drops to its minimum values (over all the stages) right after the maximum changes in target values, while MGGP + LCF both with and without hypermutation (in blue and brown colours) drops at slower rate and reaches its minimum fitness values later than pure MGGP. We provide two possible explanations for this behaviour. First, the LCF weights require time for the adaptation which makes the algorithm to "lag" behind the target. Second, the algorithm with LCFs performs much better which also means that the room for improvement (after the initial drop of fitness) is much smaller, leading to later recovery from the initial drop.

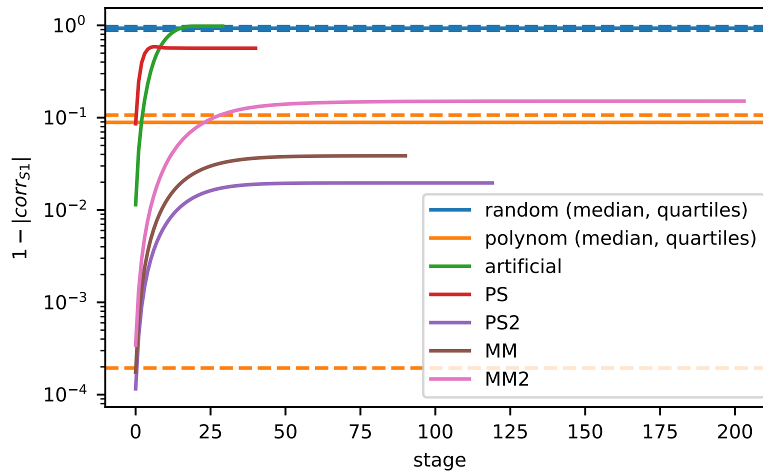


Figure 6.9. Dissimilarity of a particular stage to the first stage for each of the benchmarks.

6.6.6 Intra-stage behaviour

The primary focus of the experiments above was to examine the end-of-stage performance of the algorithms. Here we also shortly analyze the behaviour inside the individual stages. For the sake of brevity and clarity we have selected just two of the benchmarks and of those we have focused on just several stages. Figures 6.10 and 6.11 show the intra-stage behaviour in the artificial and the PS benchmarks respectively.

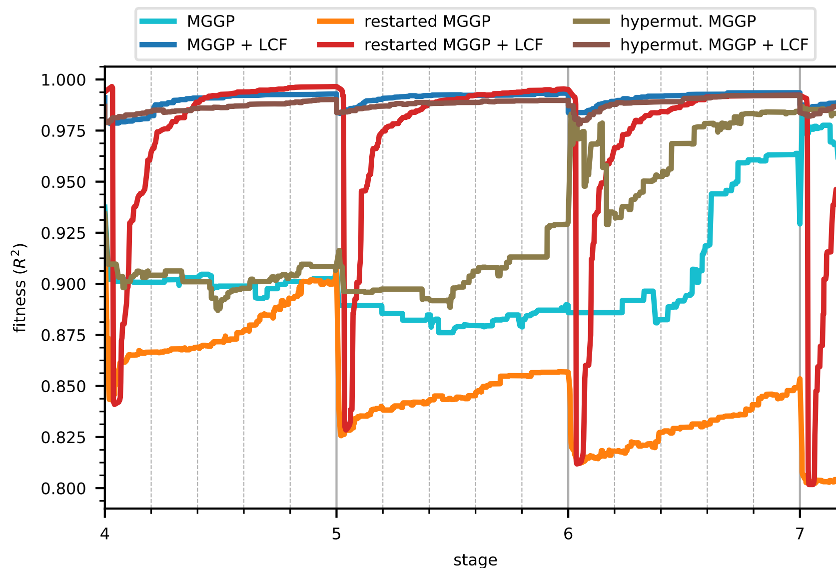


Figure 6.10. Intra-stage progress on a few selected stages on the artificial benchmark. Medians over 20 runs are plotted. Due to implementation reasons, the time resolution is one generation, which means that changes happening within one generation are not visible and only the best solution at the end of the generation is seen.

The effect of LCFs is clearly visible with the restarted algorithms. The restarted algorithm with LCFs (in red), even though the whole population is discarded, recovers very quickly, while without LCFs (in orange), the recovery is not as successful.

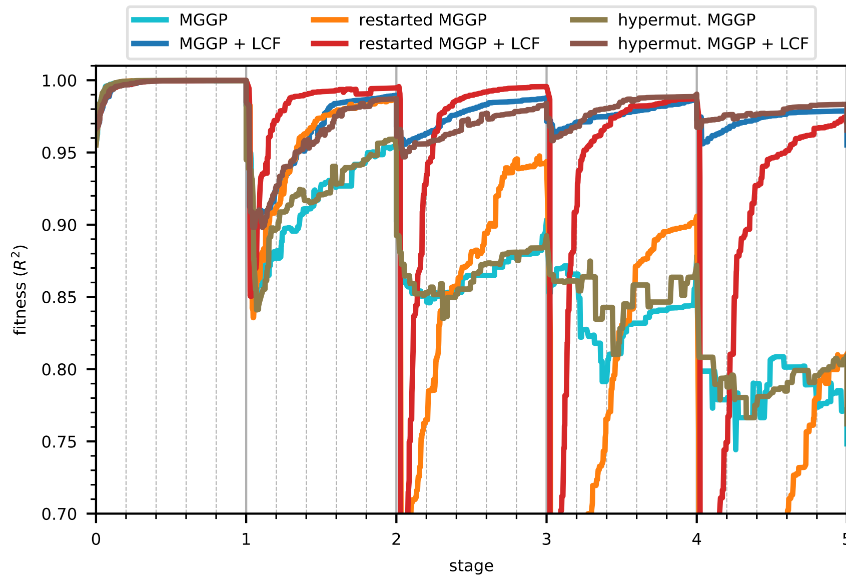


Figure 6.11. Intra-stage progress on a few selected stages on the PS benchmark. Medians over 20 runs are plotted. Due to implementation reasons, the time resolution is one generation, which means that changes happening within one generation are not visible and only the best solution at the end of the generation is seen.

In Figure 6.11 we also see that for stages 2 and 3, MGGP + LCF with restarts (red line) reached the best solutions inside the stages (despite the initial big drop in each stage), while from stage 4 on, the effect of restarting started to be disturbing, and the non-restarted algorithms provided better and more stable results.

6.7 Answers to research questions

In Section 6.5 we stated four research questions. Now, when we have performed experiments and discussed the results, it is the time to answer them. Strictly speaking, the answers are valid only in the context of the 5 benchmark problems we use. Nevertheless, we believe that they generalize to a great extent to the class of dynamic regression tasks with gradually changing target functions.

RQ1: Does MGGP with LCFs enable better target tracking compared to MGGP without them? Yes, MGGP with LCFs enables better target tracking than a pure MGGP. This is clearly seen from the presented results as MGGP + LCF is always among the best performers while MGGP is not.

RQ2: Is MGGP with LCFs better than just restarting the ordinary MGGP with each change of the target? Yes, MGGP with LCFs is better than restarting the ordinary MGGP. Restarted MGGP is even worse than pure MGGP which can be seen over all the results with the exception of the artificial benchmark where they are of comparable performance.

RQ3: Is it profitable for LCFs to retain information learned in previous stages, or is it better to start learning it from scratch after each target change? Yes, LCFs are able to utilize information found in previous stages to the advantage in future stages. This can be seen from the performance of restarted MGGP + LCF: it is worse than that

of MGGP + LCF (non-restarted) in all cases except for the artificial benchmark (for which we discussed the reasons at the beginning of Section 6.6.5).

RQ4: Is the use of LCFs better than a classic evolutionary dynamic optimisation method, hypermutation, applied to an ordinary MGGP? LCFs perform better than MGGP with hypermutation. This is true for all the benchmarks without an exception. We can see that hypermutation improves the performance of MGGP only very slightly. LCFs, on the other hand, have much bigger impact on the improvement.

Note also that hypermutation may sometimes seem to improve the performance of MGGP + LCF in certain stages. This happened, for example, in the PS2 and MM benchmarks (see Figures 6.6 and 6.7), but in either case, the difference, while visually noticeable, was not statistically significant when measured across the whole run (see Table 6.3). In the other cases, adding hypermutation to MGGP + LCF did result in an improvement.

6.8 Summary and conclusions

In this chapter, we have applied LCFs, the new type of leaf node we introduced in Chapter 5, to the task of SR where the target changes over time during the course of the evolution, making it a kind of dynamic optimization task. We have identified a subclass of dynamic SR tasks with a close relation to a practical application, reinforcement learning. An important feature of these tasks is that the modelled function changes gradually, on the contrary to other dynamic SR problems found in the literature. We have proposed five benchmarks of gradually changing target functions, the first one being a simple function that is gradually rotated, the others being V-functions of RL tasks during the course of a Value Iteration algorithm.

We have selected the “winning” configuration from the static SR benchmarks from Chapter 5 – the unsynchronized mode with gradient-based tuning – to test on the dynamic benchmarks. We have performed a proof-of-concept experiment that showed the ability of LCFs to track a simple rotating target in a vanilla GP algorithm. We have then proceeded to test the LCFs on a number of RL-based benchmarks. We have compared the two algorithms (MGGP without and with LCFs) each in three variants (no handling of changed targets, restarting and hypermutation).

The results have clearly shown that the LCF-enabled algorithm is more resistant to the gradually changing target and it is able to track it better than the algorithm without LCFs. Results of the algorithm without LCFs have shown a significant drop in the fitness when the target changes. The results have also shown that for these problems, it is beneficial to use the population from the previous stages rather than start from scratch. Using hypermutation, a classic EDO method, instead of LCFs has turned into only a slightly better performance than no change-handling method at all. The algorithm combining both the LCFs and hypermutation has been very similar in performance to using just LCFs without hypermutation so their benefits do not “add up”.

The general conclusion is that the LCFs are a beneficial addition to MGGP for dynamic SR tasks, especially when the changes are gradual. In the future, an LCF-enabled algorithm could be used as the driver for VIA in solving RL tasks, replacing the numerical model with a symbolic one, opening a range of possibilities for the RL domain.

Chapter 7
Case-based fitness prediction

In a standard SR scenario, there is a training dataset that is given to the SR algorithm to produce the model. In order to assess a quality of a candidate model (its fitness), it needs to be evaluated on this dataset. This dataset, however, can be very large. In such case, the computational demands on the algorithm can be very high.

One of the techniques to mitigate this issue is *fitness prediction* – instead of using the whole dataset to assess the fitness, use some less computationally demanding mechanism that predicts or estimates the fitness.

In this chapter we explore fitness prediction based on subsampling the training dataset. We review previous research on this topic, and we propose two simplified versions of fitness predictor search with adaptive-size predictors, which reduce the number of parameters. We compare the original and simplified methods on a number of real-world benchmarks with three different base GP algorithms.

7.1 Related work

One of the first attempts at speeding up a GP run is a random sampling [86]. The core principle of this technique is that a random subset of the training data is selected and used for fitness evaluation, resampling it periodically during the evolution. In [87] this technique was used for reducing overfitting of software quality classification models.

In [88–89], this technique was further explored and extended with the main goal of improving test-set performance / reducing overfitting. It has been shown that the extreme case of selecting just a single random datapoint each generation produces the best results for their datasets.

In [90], two new techniques, called Interleaved Sampling and Random Interleaved Sampling, further extend the random subsampling approach. In Interleaved Sampling, the evaluation switches between the full training set and a single randomly selected datapoint each generation. However, their results have shown that presenting the whole dataset has negligible effect and the performance is equivalent to using single randomly selected datapoint each generation. In Random Interleaved Sampling, the choice between a single randomly selected point and the full training set is random, but with a set probability of each choice.

In [21], a coevolutionary approach was used to evolve fitness predictors (subsets of training data in case of SR) together with the solutions. The predictors are just fixed-sized subsets of training data. The size of the predictor is a parameter that has to be determined beforehand. There are two populations used: one for the solutions, and the other one for the fitness predictors. Additionally, there is an archive of solutions used for evaluation of the predictors. The solutions are evolved with fitness being evaluated using the current best predictor. The predictors are evolved with fitness being the difference between the true fitness and predicted fitness across the solutions in the archive. Solutions that have high variance of predicted fitness across the predictor population are selected to the archive. This approach was also proposed for Cartesian

Genetic Programming (CGP) in [91] and shown to speed up the computation time by a factor of 2 to 5.

In [92–93] the predictors are coevolved such that they predict correct rank of the solutions rather than the absolute fitness value.

The disadvantage of the approach in [21, 92–93] is that the size of the predictor has to be chosen beforehand and stays fixed during the evolution. This issue has been addressed in [94], where the approach from [21] is extended by adaptively changing the size of the predictor based on the progress of the evolution. Also, the archive for predictor evolution is constructed differently: instead of selecting the solutions based on variance, the best-performing solutions are used instead. Another main difference is that the core GP algorithm used was Cartesian GP (CGP) which is quite distinct from vanilla GP. The adaptive approach was reported to have overall better success rate (ability to find a good enough solution) than other tested algorithms (no predictors, coevolved constant-size predictors, and random constant-size predictors) while consuming similar amount of CPU time or less.

7.2 Extending the adaptive-size predictor approach

From the approaches we briefly reviewed in previous section, we think that the adaptive-size predictor approach [94] is the most useful as it frees the user from the need to perform a series of experiments to determine the optimal size of the predictors. However, we also think there is still a room for improvement and extension. In this section we propose two improvements/extensions to the algorithm. For the sake of clarity, we present a high-level pseudocode of the adaptive-size predictor approach in Algorithm 7.1. Our focus is the step on line 17 – how the data points, that are used in the predictors, are selected. The original approach uses a GA and utilizes the archive A as fitness cases for the predictor candidates.

The problem of this coevolutionary approach is that in addition to the main EA, there is one extra EA that needs to be set up, i.e. its parameters (population size, crossover and mutation probabilities, tournament size, etc.) have to be determined. We propose two replacements for the inner EA: simple local search, and uniform random sampling. All other aspects of the algorithm remain the same.

7.2.1 Uniform random sampling

Randomly sampled subset of training data is not a new concept. In fact, as we have already discussed in Section 7.1, random subset selection was among the very first methods for fitness prediction (although it has not been called in such way back then). However, to the best of our knowledge, there has been no attempt at adaptive-size randomly sampled predictors. In [21] randomly sampled predictors were examined as a baseline to the coevolved predictors, but all with a fixed size of the predictor, chosen before the start of the experiment. In [94] the adaptive-size predictor approach is compared to random predictors but those are, again, fixed-sized.

We propose to take the predictor size adaptation approach and replace the inner EA with uniform random sampling. From the point of view of Algorithm 7.1, nothing changes except for what the “update active predictor” step on line 17 means: instead of running a number of generations of a GA, the new predictor is simply randomly sampled, with the number of samples determined by the currently set predictor length.

```
 1  procedure SolutionEvolution
 2      for G = 1, 2, ···                                // generations
 3          evaluate solution population
 4          S ← best solution in population
 5          if f_predictor(S) > f_predictor(S*) then
 6              store S in archive A used for predictor evaluation
 7              if f_full(S) > f_full(S*)
 8                  S* ← S
 9              end
10              update predictor length               // new length is now stored and applied only
11                                                    // when predictor is updated (line 17)
12          else if predictor length was not updated for G_update generations
13              update predictor length
14          end
15          create next generation population
16          every n-th generation
17              update active predictor
18          end
19      end
20      output: S*
21  end
```

Algorithm 7.1. High-level view on the adaptive-size predictor approach from [94] where detailed description of the algorithm can be found.

The main advantages of such approach should be:

- reduction of the number of parameters – there are no parameters related to the predictor search algorithm as there is no search (only parameters regarding the size adaptation), and
- major simplification of the algorithm – no need to implement a search algorithm as there is only random sampling which also allows to remove the solution archive completely

Whether this approach can maintain (or even improve) the performance is a question we attempt to answer in this chapter.

7.2.2 Local search for predictors

The other approach we propose is to use a local search algorithm instead of an EA to search for quality predictors. Again, the main goal is simplification of the algorithm while keeping the performance. The local search algorithm is a very basic, greedy stochastic local search, described in Algorithm 7.2. The algorithm works by randomly choosing one element in the predictor and another one not in the predictor and swapping them. If the new predictor is better it is accepted, otherwise it is rejected and the previous predictor is reverted. In either case, this process repeats a number of times and the final accepted predictor is the result of this search. To assess the quality of a predictor, the average size of absolute differences of R^2 scores of solutions in archive using the predictor vs. using the full training set is used, as can be seen on lines 9 and 15 in Algorithm 7.2.

The advantages should be similar to those of the uniform random sampling we described above, but not as pronounced, since the local search is “in between” the random

```
1  procedure PredictorLocalSearch
2      inputs:
4          ℙ,                          // initial predictor
5          𝕏,                          // the whole training dataset
6          A,                          // archive of individuals
7          n_max                       // number of iterations
8      ℝ ← 𝕏 \ ℙ                       // the "rest" of datapoints (not in predictor)
9      q ← 1/|A| Σ_{i∈A} |f_ℙ(i) − f_𝕏(i)|
10     for n = 1, 2, ..., n_max do
11         a ← ℛ(ℙ)
12         b ← ℛ(ℝ)
13         ℙ' ← (ℙ \ {a}) ∪ {b}
14         ℝ' ← (ℝ \ {b}) ∪ {a}
15         q' ← 1/|A| Σ_{i∈A} |f_ℙ(i) − f_𝕏(i)|
16         if q' < q then
17             q ← q'
18             ℙ ← ℙ'
19             ℝ ← ℝ'
20         end
21     end
22     output: ℙ
23 end
```

Algorithm 7.2. Pseudocode for the predictor local search algorithm, where $f_{\mathbb{S}}(a)$ is the fitness (R^2 score) of individual a using the set of datapoints $\mathbb{S}$ and $\mathcal{R}(\mathbb{S})$ represents an element uniformly randomly chosen from the set $\mathbb{S}$.

sampling and GA in terms of complexity: It needs some parameters (mainly the number of iterations per predictor update) but not as much as a GA, and it is more complicated algorithm than simple random sampling but not as complicated as GA.

The effectiveness of this approach is to be answered in this chapter as well.

7.3 Research questions

Having proposed our modifications to the adaptive-size predictor approach, we ask the following research questions:

- RQ1: Can uniform random sampling replace the inner GA as the algorithm for finding suitable fitness predictors, while keeping or improving the performance?
- RQ2: Can local search replace the inner GA as the algorithm for finding suitable fitness predictors, while keeping or improving the performance?
- RQ3: Are fitness predictors beneficial when LCFs (see Chapter 5) are used?

7.4 Experimental evaluation

7.4.1 Base algorithms

To assess the benefit of fitness prediction in different environments, we use three base GP algorithms to which the fitness prediction is added as an extension.

GP This is a vanilla GP algorithm, i.e. the individuals are single trees, and no linear fitting is performed.

MGGP This is a standard MGGP algorithm, i.e. the individuals are composed of one or more trees, and they are linearly fitted to the training data. This is the same algorithm that has been used in Chapter 5 as the base for extension with LCFs.

MGGP+LCF This is an MGGP algorithm with the LCF extension that has been introduced in Chapter 5. Gradient-based tuning is used to find the LCF parameters and the *unsynchronized* mode is used, see Section 5.1 for details.

7.4.2 Fitness prediction methods

We examine three fitness prediction approaches plus a control configuration where no fitness prediction is employed.

Coevolution (CE) This is a 1:1 reimplementation of the approach used in [94], except the base algorithm is not CGP but one of the above-mentioned. In short, in this approach, the predictors are co-evolved alongside the main evolution, and the size of the predictor is adapted over the course of the run of the algorithm.

Random (R) This is the approach we proposed in Section 7.2.1. Here, the points that form the predictor are chosen randomly (without repetition). However, the same predictor size adaptation mechanism as in [94] is used.

Local search (LS) This is the approach we proposed in Section 7.2.2. Here, the points that form the predictor are found using a simple greedy local search. The size of the predictor, though, is also determined using the mechanism from [94].

None For proper assessment of the benefit of using fitness prediction, we also include the “naked” base algorithms without any fitness prediction at all, i.e. the full training set is used for all evaluations of all individuals.

7.4.3 Benchmarks

We have selected 6 datasets for evaluation of the performance of the algorithms. The datasets are summarized in Table 7.1 and further described below.

name	# of features	# of datapoints		source
		train	test	
ASN	5	1052	451	[95]
ParkinsonMotor	18	4112	1763	[96, 95]
ParkinsonTotal	18	4112	1763	[96, 95]
puma8NH	8	5734	2458	[97]
SupCon	81	14884	6379	[98, 95]
WEC-A	48	50399	21600	[95]

Table 7.1. Summary of datasets used for benchmarks of fitness prediction algorithms.

ASN Full name is Airfoil Self-Noise and it is a dataset with data from testing airfoils in a wind tunnel. The goal is to predict sound pressure levels for these airfoils. There are 1503 datapoints in total and we have split them randomly 70:30 into training and testing subsets.

ParkinsonMotor, ParkinsonTotal In these datasets, the goal is to predict motor and total UPDRS score based on features extracted from voice measurements of 42 early-stage Parkinson's disease patients. The measurements come from a telemonitoring device for remote symptom progression monitoring. Each data point corresponds to one voice measurement. There are 5875 datapoints in total and we have split them randomly 70:30 into training and testing subsets.

puma8NH The data in this dataset come from a realistic simulation of dynamic of an Unimation Puma 560 robot arm. The goal is to predict an angular acceleration of one of the robot arm's links, given angular positions, velocities and torques of the arm. There are 8192 datapoints in total and we have split them randomly 70:30 into training and testing subsets.

SupCon This dataset contains features of superconductor materials along with their critical temperatures which is the prediction target. There are 21263 datapoints in total and we have split them randomly 70:30 into training and testing subsets.

WEC-A This dataset contains data about positions and absorbed power of a number of Wave Energy Converters near Adelaide, Australia. The goal is to predict the total output power based on the positions and absorbed powers. The full dataset consists of four locations (Sydney, Adelaide, Perth and Tasmania) of which we have chosen only Adelaide. There are 71999 datapoints in total and we have split them randomly 70:30 into training and testing subsets.

7.4.4 Performance metrics

There are two performance metrics that we measure:

- end-of-run R^2 score on testing set (EoR-R^2 for short), and
- relative area under curve (RAUC for short) of the evolution plot.

Both metrics are illustrated in Figure 7.1.

The value of EoR-R^2 for one run of an algorithm is the testing set R^2 score of the model that was overall best during the evolution with respect to the training set. The value of RAUC for one run of an algorithm is obtained with this procedure: whenever a new best model (with respect to the training set) is found during the evolution, its R^2 score on the testing set as well as the wall-clock time since the start of the run are recorded. The RAUC is then calculated as

$$RAUC = \frac{\left(\sum_{k=1}^{N-1} R_k^2 \cdot (t_{k+1} - t_k)\right) + R_N^2 \cdot (T - t_N)}{T}$$

where R_k^2 is the test-set R^2 score of the k-th model found, t_k is the time of that model being found, N is the total number of models found, and T is the total time given to the algorithm which was 900 seconds.

For the purposes of calculating the RAUC metric, models that have *training* $R^2 < 0$ are excluded so that extremely bad models (typically at the start of the run) don't pollute the metric (and also because in such case a simple constant model equal to the mean of training data could be used instead).

Since the sum is divided by the total running time T, the value of $RAUC = 1$ means that the algorithm has found a perfect model right at the very start of the run, the value of $RAUC = 0$ means that the algorithm has not been able to produce any model

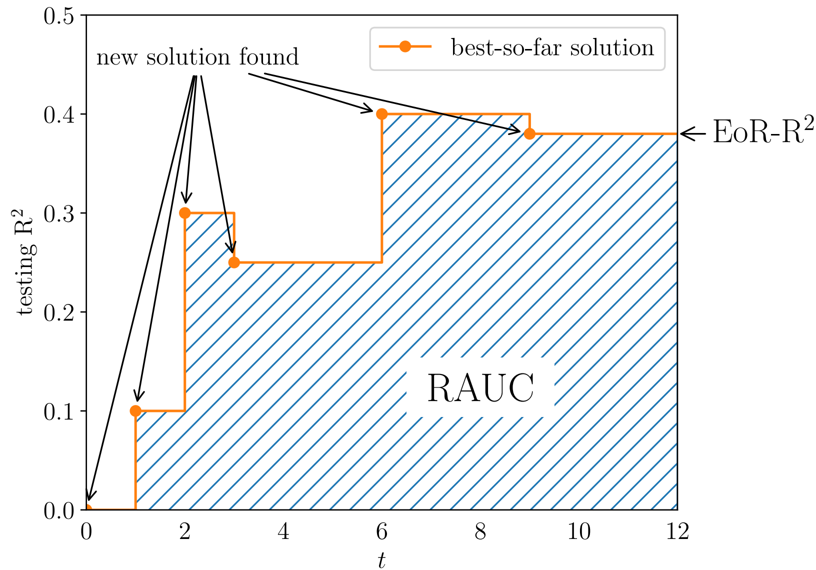
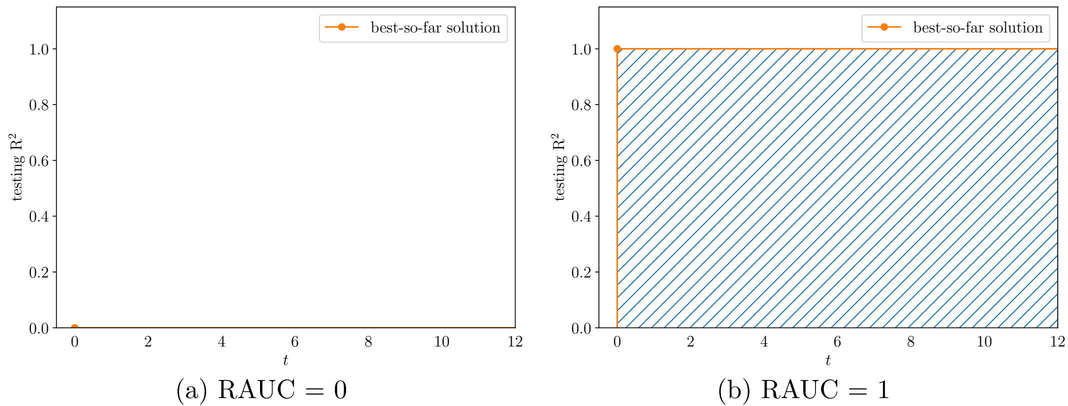


Figure 7.1. Illustration of the EoR-R^2 and RAUC performance metrics. EoR-R^2 is the test set R^2 of the last model (the best on training data), RAUC corresponds to the area under the plot of test set R^2 scores of the best solution, divided by the total time. In this case, EoR-$R^2 = 0.38$, and RAUC $= \frac{0 \cdot 1 + 0.1 \cdot 1 + 0.3 \cdot 1 + 0.25 \cdot 3 + 0.4 \cdot 3 + 0.38 \cdot 3}{12} \approx 0.29$. Note that the "best-so-far" relates to training performance which is the reason the best-so-far solution can get worse from the testing point of view.



(a) RAUC = 0 (b) RAUC = 1

Figure 7.2. Illustration of the edge cases of the RAUC metric. In (a) RAUC = 0 because the only solution found had $R^2 = 0$. In (b) RAUC = 1 because a solution with $R^2 = 1$ was found at the very start of the algorithm.

that would be better than a simple constant model of mean of training targets values[1]. These edge case situations are illustrated in Figure 7.2.

Values in between 0 and 1 mean that the algorithm found a non-perfect model(s) and/or it found them later in the evolution[2].

[1] This follows from the definition of $R^2 = 1 - \frac{\sum_{i=1}^{N}(f(\mathbf{x}_i) - y_i)^2}{\sum_{i=1}^{N}(\bar{y} - y_i)^2}$ where f is the evaluated model. If $f(\mathbf{x}) = \bar{y}$ (or worse) then $R^2 = 0$ (or less), therefore it is counted towards RAUC as 0. If no better model is found, RAUC stays at 0.

[2] Another interpretation of this metric is that it is the test-set R^2 averaged over the run of the evolution.

A third, supplementary metric is the size of the models in number of nodes. All three metrics – EoR-R^2, RAUC, and No. of nodes – are aggregated over all runs of an algorithm and statistically compared.

7.4.5 Algorithm parameters

The parameters of the base algorithms (i.e. GP, MGGP, MGGP+LCF) are depicted in Table 7.2. Note that parameters that are common for multiple algorithms are set to the same value for all of them.

parameter	value	applies for		
		GP	MGGP	MGGP+LCF
pop. size	300	✓	✓	✓
tournament size	12	✓	✓	✓
# of elites	30	✓	✓	✓
max. depth	10	✓	✓	✓
mut. prob.	0.2	✓	✓	✓
cross. prob.	0.7	✓	✓	✓
const. mut. prob.	0.3	✓	✓	✓
const. mut. σ	10	✓	✓	✓
# of genes	5		✓	✓
high-level cross. prob.	0.2		✓	✓
high-level cross. rate	0.5		✓	✓
# of backprop. steps	3			✓
weights mut. prob.	0.05			✓
weights mut. σ	3			✓
function set	$a+b, a-b, a \cdot b, \frac{a}{b}, a^2, a^3, a^4,$ $\sin a, \cos a, \exp a, \log a$	✓	✓	✓

Table 7.2. Parameters of base algorithms. The parameter “const. mut. prob.” is the probability that a mutation (if a mutation already takes place) is a numerical constant leaf node mutation. Otherwise, the mutation is a subtree mutation.

The other part of the settings are the fitness prediction parameters. Since each fitness prediction approach brings with itself a number of parameters, we performed a grid search over these parameters with grid points that we considered as sensible values. The parameters and their corresponding grid values are listed in Table 7.3.

We allowed only a single value for some of the parameters. Parameters $|P_{pred}|$, $|T_{pred}|$, $p_{pred,cross}$, and $p_{pred,mut}$ were simply taken from [94] as these apply only for the CE configurations which uses the same inner GA. Parameter l_{min} was set as the lowest number where the linear regression task in MGGP algorithm can be fully determined – there are 5 genes (see Table 7.2) and an intercept. Parameter u was determined based on preliminary experiments.

The grid search was performed on datasets ASN (as a representative of a “small” dataset) and SupCon (as a representative of a “big” dataset) by running each configuration (combination of base algorithm and prediction method with one set of parameters) 10 times. Then, for each base algorithm and each fitness prediction approach, two configurations are selected based on the two performance metrics we described in Section 7.4.6: the best w.r.t. the end-of-run test-set R^2, and the best w.r.t. the relative area under curve. The first criterion should provide a configuration that produces good

description	symbol	values	applies for CE	LS	R
# of generations between predictor updates	i	20, 40, 60, 80	✓	✓	✓
# of inner evolution generations	g	1, 10, 20	✓		
# of local search iterations	r	100, 800, 1500		✓	
predictor population size	$\|P_{pred}\|$	32	✓		
predictor tournament size	$\|T_{pred}\|$	2	✓		
predictor crossover prob.	$p_{pred,cross}$	0.8	✓		
predictor mutation prob.	$p_{pred,mut}$	0.2	✓		
size of archive for predictor evaluation	$\|A\|$	75, 300, 600	✓	✓	
minimum length of a predictor	l_{min}	6	✓	✓	✓
initial length of the predictor as fraction of all data	l_{init}	0.05, 0.5	✓	✓	✓
maximum # of generations after which the desired length of the predictor is recalculated	G_{update}	8	✓	✓	✓
see [94], Section 3.1.2, step 3	I_{thr}	0.03, 0.1	✓	✓	✓

Table 7.3. Grid search points for parameters of fitness prediction methods.

results at the end of the run, while the second criterion should provide a configuration that produces good results fast. The “best” configuration, according to the given criterion, is selected by putting all runs of all tested configurations together and ranking them, then the ranks are summed across both datasets, and, finally, the configuration with the lowest sum of ranks of its runs is selected. In the case when the same configuration is the best in both metrics (for one fitness prediction approach), in order to keep two configurations per fitness prediction approach, the second-best configuration from the RAUC point of view is taken as the second one. The control configurations with no fitness prediction obviously did not go through the grid search as they have no fitness prediction parameters to be determined.

Selected configurations. The configurations selected using this process for GP, MGGP and MGGP+LCF are summarized in Table 7.4. For the sake of brevity, we will distinguish these configurations using a “codename” in the form

X-Y-Z

where “X” refers to the base algorithm (GP, MGGP, or MGGP+LCF), “Y” refers to the fitness prediction method used (CE, LS, or R), and “Z” refers to the metric based on which was the particular configuration was selected (“A” for RAUC, “E” for EoR-R^2).

7.4.6 Testing methodology

Each combination of base algorithm (GP, MGGP, MGGP+LCF), fitness prediction (coevolution, local search, random, and no prediction), and set of parameters (best by RAUC, best by EoR-R^2; only one set for base algorithms without prediction) i.e. 21 distinct algorithm configurations, is run 20 times independently on each dataset. The performance metrics are collected for each configuration independently, aggregated over the 20 runs and statistically compared.

	parameter	*-CE-A	*-LS-A	*-R-A	*-CE-E	*-LS-E	*-R-E
GP-*	i	40	80	20	20	20	20
	r	10	100	–	1	800	–
	$\|A\|$	600	300	–	75	75	–
	l_{min}	6	6	6	6	6	6
	l_{init}	0.05	0.5	0.05	0.5	0.05	0.05
	u	8	8	8	8	8	8
	I_{thr}	0.03	0.03	0.1	0.1	0.1	0.03
MGGP-*	i	20	40	60	60	40	60
	r	20	800	–	1	800	–
	$\|A\|$	75	300	–	300	75	–
	l_{min}	6	6	6	6	6	6
	l_{init}	0.05	0.05	0.5	0.05	0.5	0.5
	u	8	8	8	8	8	8
	I_{thr}	0.03	0.03	0.1	0.1	0.03	0.03
MGGP+LCF-*	i	20	20	20	20	40	20
	r	20	100	–	10	1500	–
	$\|A\|$	300	75	–	300	75	–
	l_{min}	6	6	6	6	6	6
	l_{init}	0.05	0.05	0.05	0.05	0.5	0.5
	u	8	8	8	8	8	8
	I_{thr}	0.1	0.1	0.03	0.03	0.1	0.1

Table 7.4. Selected configurations of fitness prediction methods for GP base algorithm. *Note:* configurations GP-LS-A, MGGP-R-A and MGGP-CE-A were not the best configurations for given base algorithm, fitness prediction approach and performance metric but the second-best ones because the best ones were identical to the best ones in the other metric.

All runs are performed on the MetaCentrum cluster such that only machines of identical configuration are utilized.

7.4.7 Results and discussion

Now we present the results of the selected configurations on all datasets. Tables 7.5 and 7.6 show the median values of EoR-R^2 and RAUC metrics respectively. Table 7.7 then shows the median number of nodes of the final model. Detailed view in form of boxplots can be found in Appendix C.1.

Table 7.5 shows that, except for GP-CE-A on the ParkinsonTotal dataset, no fitness prediction configuration was significantly better or worse than the base algorithm, when EoR-R^2 is considered. From the RAUC point of view, as shown in Table 7.6, there are more differences, especially with the MGGP+LCF-based algorithms on the WEC-A dataset, but the overall picture is similar – mostly no difference from the base algorithms. When the model size (number of nodes) is considered, which is depicted in Table 7.7, it can be said that fitness prediction did not improve in this regard and multiple times produced even more complex models than the base algorithm.

Regarding the reason for why the MGGP+LCF-based algorithms actually worsened their RAUC performance over the base algorithm on the WEC-A dataset, we propose the following explanation. The dataset seems to be “easy” to solve as all algorithms and fitness prediction configurations (including the configurations with no fitness prediction)

	dataset	no pred.	*-CE-A	*-CE-E	*-LS-A	*-LS-E	*-R-A	*-R-E
GP-*	ASN	0.471	0.459	0.528	0.482	0.500	0.416	0.411
	ParkinsonMotor	0.368	0.470	0.409	0.350	0.379	0.417	0.423
	ParkinsonTotal	0.359	0.382	0.306	0.295	0.372	0.450	0.432
	puma8NH	0.673	0.677	0.676	0.670	0.666	0.671	0.677
	SupCon	0.588	0.615	0.612	0.573	0.618	0.626	0.612
	WEC-A	>0.999	>0.999	>0.999	>0.999	>0.999	>0.999	>0.999
	total (+|=|-)	–	4|1|1	4|1|1	1|1|4	4|1|1	3|1|2	4|1|1
	total sig. (+|=|-)	–	0|6|0	0|6|0	0|6|0	0|6|0	0|6|0	0|6|0
MGGP-*	ASN	0.797	0.806	0.810	0.808	0.786	0.805	0.808
	ParkinsonMotor	0.728	0.734	0.737	0.740	0.723	0.752	0.740
	ParkinsonTotal	0.701	**0.743**	0.716	0.712	0.733	0.713	0.726
	puma8NH	0.683	0.681	0.682	0.684	0.684	0.683	0.683
	SupCon	0.753	0.758	0.760	0.754	0.761	0.758	0.759
	WEC-A	>0.999	>0.999	>0.999	0.999	>0.999	>0.999	>0.999
	total (+|=|-)	–	4|1|1	4|1|1	5|0|1	3|1|2	4|2|0	4|2|0
	total sig. (+|=|-)	–	1|5|0	0|6|0	0|6|0	0|6|0	0|6|0	0|6|0
MGGP+LCF-*	ASN	0.760	0.782	0.786	0.783	0.771	0.789	0.788
	ParkinsonMotor	0.448	0.481	0.472	0.546	0.491	0.466	0.538
	ParkinsonTotal	0.430	0.442	0.500	0.551	0.428	0.613	0.455
	puma8NH	0.673	0.673	0.670	0.669	0.675	0.671	0.674
	SupCon	0.718	0.742	0.749	0.728	0.734	0.724	0.725
	WEC-A	>0.999	>0.999	>0.999	>0.999	>0.999	>0.999	>0.999
	total (+|=|-)	–	4|2|0	4|1|1	4|1|1	4|1|1	4|1|1	5|1|0
	total sig. (+|=|-)	–	0|6|0	0|6|0	0|6|0	0|6|0	0|6|0	0|6|0

Table 7.5. Aggregated results of the fitness prediction algorithms on all datasets as viewed by the EoR-R^2. The "no pred." column shows the performance of the base algorithm without any prediction. The values are medians for the particular algorithm configuration and dataset over 20 independent runs. Values in green/red indicate that the median is better/worse than that of the base algorithm without prediction. Values set in bold indicate that their distributions also differ according to the two-sided MWUT with significance level $\alpha = 0.01$. The rows "total (+|=|-)" show the total number of times the particular configuration was better | equal to | worse than the base algorithm without prediction when comparing just the medians. The rows "total sig. (+|=|-)" are similar, but only the significant differences are considered better/worse.

achieved almost perfect median of nearly 1.0, as seen in Table 7.5. Also, MGGP+LCF uses only 5 nodes to do that, as seen in Table 7.7. This means that it heavily utilizes the LCF nodes as those are counted as a single node, and indeed, an inspection of the models shows 5-gene models where the genes are each just a single LCF node[3]. This also means that the LCF nodes have to be tuned properly. Since the fitness prediction makes the algorithm use less datapoints, the signal received from the error backpropagation

[3] This means that the model is linear. If the MGGP algorithm was allowed to use (at least) as many genes as there are features in the WEC-A dataset (48) it would be able to directly produce optimal linear model as it utilizes linear regression. However, the maximum number of genes is just 5 so this is not possible. With the LCFs, however, the algorithm can "pack" the features into the LCF nodes and optimize the parameters via gradient descent.

	dataset	no pred.	*-CE-A	*-CE-E	*-LS-A	*-LS-E	*-R-A	*-R-E
GP-*	ASN	0.340	0.330	0.389	0.379	0.300	0.280	0.340
	ParkinsonMotor	0.272	0.350	0.354	0.238	0.245	0.335	0.347
	ParkinsonTotal	0.272	0.267	0.213	0.250	0.205	0.354	0.319
	puma8NH	0.665	0.664	0.668	0.661	**0.645**	0.662	0.665
	SupCon	0.524	0.555	0.557	0.537	0.568	0.587	0.571
	WEC-A	0.859	0.939	0.890	0.868	0.899	0.951	0.942
	total (+|=|-)	–	3|0|3	5|0|1	3|0|3	2|0|4	4|0|2	4|2|0
	total sig. (+|=|-)	–	0|6|0	0|6|0	0|6|0	0|5|1	0|6|0	0|6|0
MGGP-*	ASN	0.760	0.778	0.748	0.744	0.752	0.775	0.782
	ParkinsonMotor	0.652	0.666	0.654	0.635	0.646	0.673	0.666
	ParkinsonTotal	0.651	0.688	0.664	0.636	0.654	0.648	0.668
	puma8NH	0.676	0.672	0.674	0.669	0.677	0.677	0.677
	SupCon	0.720	0.731	0.736	0.734	0.736	0.727	0.725
	WEC-A	0.954	**0.971**	**0.976**	0.968	0.951	0.951	0.958
	total (+|=|-)	–	5|0|1	4|0|2	2|0|4	2|0|4	4|0|2	6|0|0
	total sig. (+|=|-)	–	1|5|0	1|5|0	0|6|0	0|6|0	0|6|0	0|6|0
MGGP+LCF-*	ASN	0.725	0.743	0.739	0.710	0.728	0.738	0.745
	ParkinsonMotor	0.396	0.376	0.367	0.468	0.429	0.396	0.380
	ParkinsonTotal	0.358	0.348	0.373	0.465	0.340	**0.516**	0.338
	puma8NH	0.657	0.653	0.651	0.650	0.661	0.651	0.663
	SupCon	0.688	**0.728**	**0.730**	**0.715**	0.704	**0.708**	0.687
	WEC-A	0.984	**0.981**	**0.981**	**0.992**	**0.961**	**0.991**	**0.954**
	total (+|=|-)	–	2|0|4	3|0|3	4|0|2	4|0|2	5|1|0	2|0|4
	total sig. (+|=|-)	–	1|4|1	1|4|1	2|4|0	0|5|1	3|3|0	0|5|1

Table 7.6. Aggregated results of the fitness prediction algorithms on all datasets using the RAUC performance characteristic. The "no pred." column shows the performance of the base algorithm without any prediction. The values are medians for the particular algorithm configurations and dataset over 20 independent runs. Values in green/red indicate that the median is better/worse than that of the base algorithm without prediction. Values set in bold indicate that their distributions also differ according to the two-sided MWUT with significance level $\alpha = 0.01$. The rows "total (+|=|-)" show the total number of times the particular configuration was better | equal to | worse than the base algorithm without prediction when comparing just the medians. The rows "total sig. (+|=|-)" are similar, but only the significant differences are considered better/worse.

is weaker than in the case of using the full dataset. As for why this does not happen for the other datasets, we think that it is because they require more complex models where the LCF nodes do not play as important role. Also note that even though the difference is statistically significant, it is not very big in terms of absolute values.

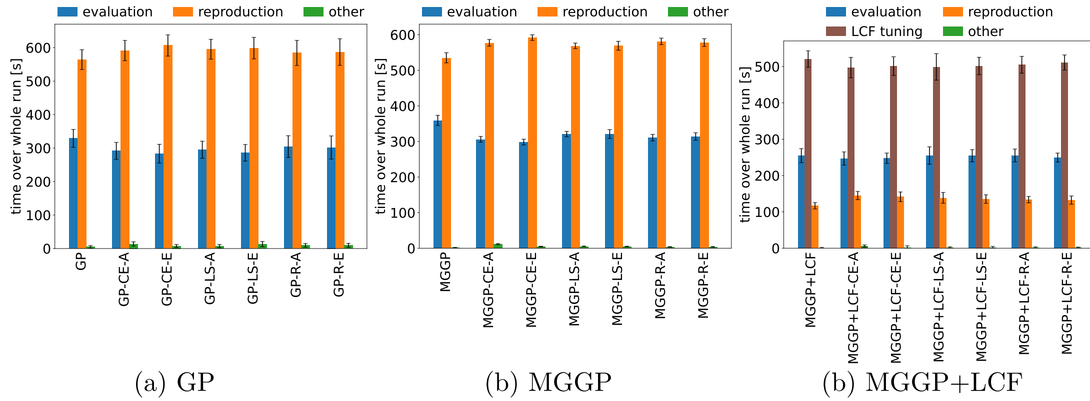
Overall, using fitness prediction should help the algorithms work faster and therefore achieve better results in the end, or achieve the same results sooner in the run, but that is not what can be observed in the presented results, except for a few cases. This is somewhat unexpected, since previous research has shown that fitness prediction improves the underlying algorithms. We have three hypotheses that try to explain the cause of this discrepancy.

	dataset	no pred.	*-CE-A	*-CE-E	*-LS-A	*-LS-E	*-R-A	*-R-E
GP-*	ASN	72.5	63.5	84.5	82.5	60.5	68.5	58.5
	ParkinsonMotor	75.5	99	110.5	90	95.5	90.5	96
	ParkinsonTotal	75	84.5	97	88	95	110.5	100.5
	puma8NH	60.5	87.5	74.5	84	90.5	81.5	85.5
	SupCon	66	68	86.5	68	72	63.5	83
	WEC-A	34.5	38	44.5	35	31	36.5	38
	total (+|=|-)	–	0|5|1	0|6|0	0|5|1	0|5|1	0|5|1	0|5|1
MGGP-*	ASN	155.5	137	144	130	135	144.5	136.5
	ParkinsonMotor	79	102	108	110.5	99	102	89.5
	ParkinsonTotal	93.5	109	99.5	98	100.5	96	104
	puma8NH	72.5	75.5	61.5	83	90.5	71	73
	SupCon	56.5	109.5	77.5	77	82.5	80	83
	WEC-A	39.5	42	37.5	57	37.5	48	49
	total (+|=|-)	–	0|5|1	0|5|1	0|5|1	0|5|1	0|5|1	0|5|1
MGGP+LCF-*	ASN	44	42.5	43.5	48	37.5	40	41.5
	ParkinsonMotor	16.5	28	19	34.5	18.5	29.5	17.5
	ParkinsonTotal	16	31	32	32	26	27.5	26.5
	puma8NH	13	21	18	30	21	22.5	18
	SupCon	5.5	7.5	7	7.5	12	7.5	6
	WEC-A	5	5	5	5	5	5	5
	total (+|=|-)	–	0|6|0	0|5|1	0|3|3	0|6|0	0|5|1	0|6|0

Table 7.7. Aggregated results of the fitness prediction algorithms on all datasets in terms of number of nodes of the final model. The "no pred." column shows the value for the base algorithm without any prediction. The values are medians for the particular algorithm configurations and dataset over 20 independent runs. Values highlighted in red/green indicate that the median is worse/better than that of the base algorithm without prediction, and their distributions differ according to the two-sided MWUT with significance level $\alpha = 0.01$. A non-highlighted value means that no difference in distributions was found according to this test. This result is summarized in the row "total (+|=|-)" which indicates the number of times the particular configuration was better | indifferent to | worse than the base algorithm without prediction.

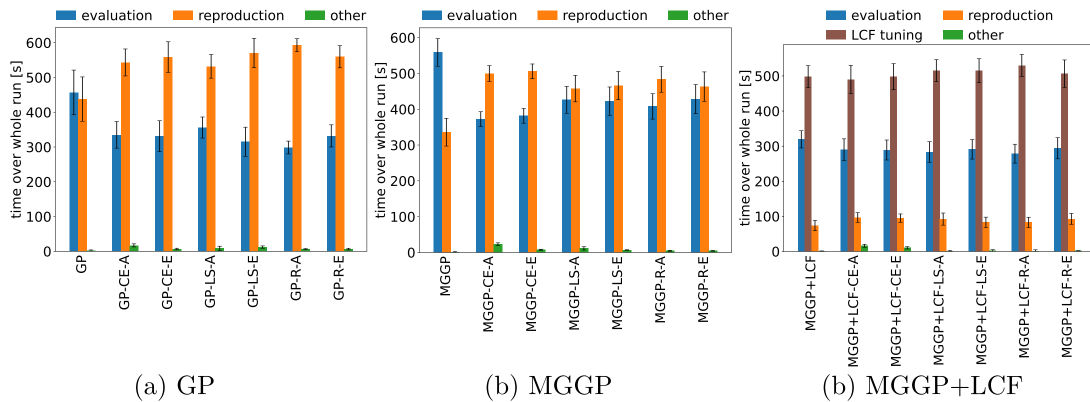
H1: The fitness evaluation is not the bottleneck of the algorithm. Our implementation is written in Python with the actual evaluation using the numpy library [99]. Since Python is an interpreted language while numpy has a highly efficient pre-compiled core utilizing OpenBLAS[4] [100], the evaluation is significantly faster than the rest of the algorithm and therefore the benefit of speeding it up (by evaluating less data) is not as pronounced as in an implementation that is written entirely in a compiled language like C, where the evaluation could actually be the bottleneck.

[4] In our environment.



(a) GP (b) MGGP (b) MGGP+LCF

Figure 7.3. Timing of algorithms on the ASN dataset. The columns show mean time spent in the corresponding part of the algorithm, the error bars show standard deviation.



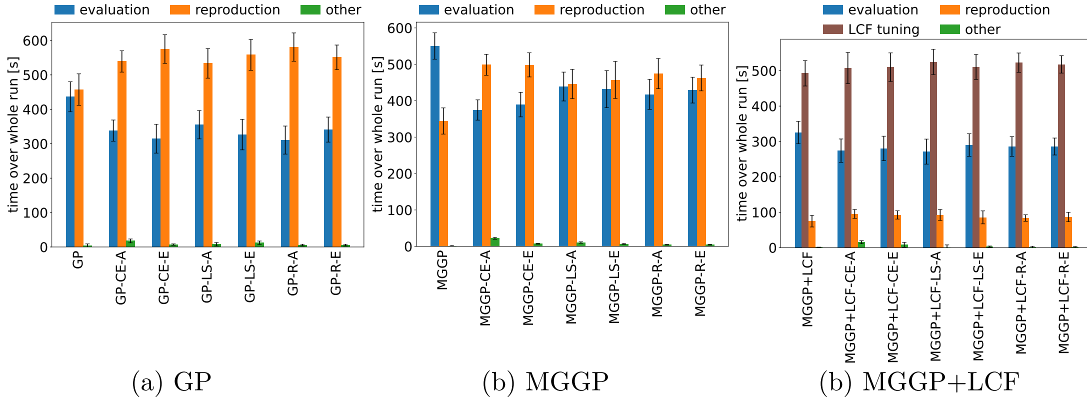
(a) GP (b) MGGP (b) MGGP+LCF

Figure 7.4. Timing of algorithms on the ParkinsonMotor dataset. The columns show mean time spent in the corresponding part of the algorithm, the error bars show standard deviation.

To get some insight into this issue, we analyze the timing of the algorithms. Figures 7.3 through 7.8 show how much time is spent in evaluation[5] of the data, tuning the LCF parameters[6], reproduction[7], and other parts[8].

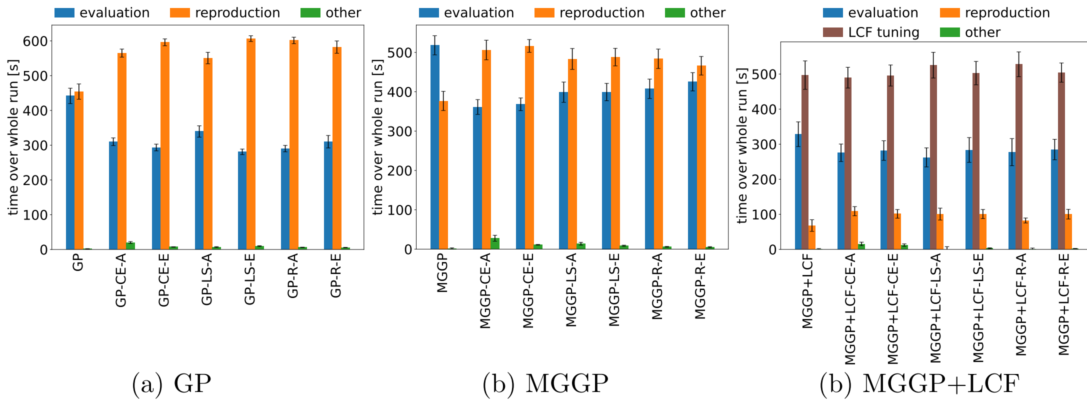
We can see that on ASN (which is the smallest dataset in terms of number of datapoints), both GP and MGGP spend the majority of time in reproduction. Introducing fitness prediction changes this ratio only slightly. On the other hand, MGGP+LCF spends the majority of time in gradient-based tuning of LCF parameters. Again, introducing fitness prediction changes this only slightly. On ParkinsonMotor, ParkinsonTotal, and puma8NH (the "mid-sized" datasets), the picture is similar except that for GP, the ratio is roughly 1:1, and for MGGP, the majority is spent in evaluation. Introducing fitness prediction has stronger impact than it had on the ASN dataset. On the two largest datasets – SupCon and WEC-A – the majority of time is spent in evaluation for GP and MGGP, and fitness prediction lessens the difference. In MGGP+LCF, the most time is spent roughly equally in evaluation and LCF tuning, and fitness prediction decreases the evaluation time in favour of the LCF tuning.

[5] For MGGP and MGGP+LCF, evaluation includes also the linear regression on the outputs of the genes.
[6] Only for MGGP+LCF.
[7] Includes selection, crossover and mutation.
[8] Includes selection of elites, various bookkeeping routines, and also the procedure that creates the fitness predictors, i.e. coevolution, local search or random sampling.



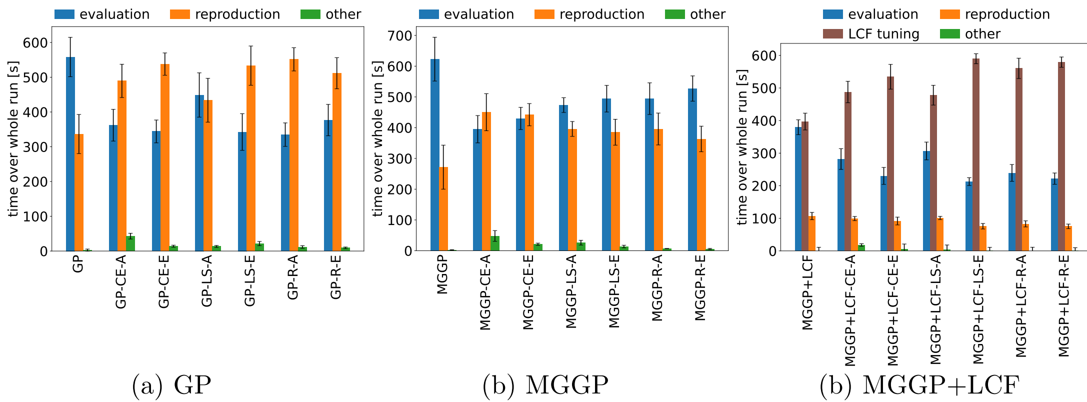
(a) GP (b) MGGP (b) MGGP+LCF

Figure 7.5. Timing of algorithms on the ParkinsonTotal dataset. The columns show mean time spent in the corresponding part of the algorithm, the error bars show standard deviation.



(a) GP (b) MGGP (b) MGGP+LCF

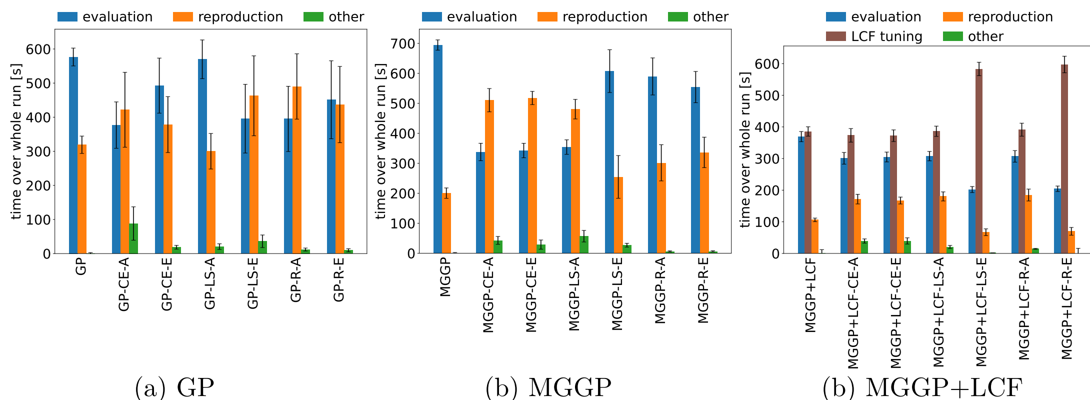
Figure 7.6. Timing of algorithms on the puma8NH dataset. The columns show mean time spent in the corresponding part of the algorithm, the error bars show standard deviation.



(a) GP (b) MGGP (b) MGGP+LCF

Figure 7.7. Timing of algorithms on the SupCon dataset. The columns show mean time spent in the corresponding part of the algorithm, the error bars show standard deviation.

We can definitely see that, at least for GP and MGGP, reproduction takes a significant amount of time of the algorithm which might indicate that this hypothesis – that evaluation is not the bottleneck of the algorithm and therefore speeding it up does not help – may have some merit. This hypothesis is testable – although we cannot just make the non-evaluation part of the algorithm faster, we can make the evaluation part slower by simply introducing a slight artificial delay for each datapoint being evaluated.



(a) GP (b) MGGP (b) MGGP+LCF

Figure 7.8. Timing of algorithms on the WEC-A dataset. The columns show mean time spent in the corresponding part of the algorithm, the error bars show standard deviation.

H2: The base algorithm itself (i.e. GP and MGGP in this case) is more computationally demanding than CGP used in [94]. This is a reasonable hypothesis since CGP does not require any tree structure manipulation, it only has a fixed-sized grid of functions where the interconnections and the individual functions are manipulated.

However, this hypothesis is difficult to verify since it would require comparison with the algorithms form the referenced papers, implementations of which are not available. Also, in [21] a coevolutionary approach to fitness prediction was used with a classic tree-based GP[9], and was reported to be significantly better than the algorithm without fitness prediction. This makes the hypothesis less likely to be true.

H3: Previous fitness predictor research is not entirely compatible with the experiments performed here. In [6, 94], the SR benchmarks used for evaluation of the algorithms were artificial, one-dimensional test problems only. In those experiments, the test problems were such that they could be, in principle, modelled perfectly because the function sets used by the algorithms had all the building blocks needed to reconstruct the sought functions. This might not be the case for real-world datasets. Also, the processes themselves, that gave rise to the real-world data, are likely more complex than the artificial benchmarks. It is possible that fitness prediction is simply not as effective in such situation.

This hypothesis is plausible and verifiable by simply conducting experiments on these test problems as well. However, if H1 is true, that effect could contaminate the results of the verification experiments, leading to false results. Beside this major point, we also think that such simple datasets are not very interesting and have only limited potential for showing the performance of the algorithms.

We think that H1 is the most likely explanation and also the most fundamental one, and we have means to verify it. Therefore, in the rest of this chapter, we are going to focus on eliminating the effect described in H1.

7.5 Experiments simulating more time-demanding evaluation

The results presented in the previous section show only a little difference, between the base algorithm and its variants with fitness prediction. We proposed a hypothesis that

[9] The implementation is not publicly available and cannot be analyzed, though.

this is caused by the fact that the evaluation is actually not the most time consuming part of the algorithm, due to our implementation being written in Python with the exception of the evaluation itself, which utilizes a highly efficient numpy library.

In order to test this hypothesis, we perform identical experiments, but we modify the algorithm by introducing an artificial delay for each datapoint being evaluated. In our environment, evaluation of a single datapoint takes, on average[10], 1.18 μs. We decided to introduce a delay of double this time, i.e. 2.36 μs, therefore the evaluation should take roughly three times more time than with the regular algorithm.[11]

7.5.1 Results and discussion

As in the original experiments, we present the results in the same form. Tables 7.8 and 7.9 show the median values of EoR test-set R^2 and RAUC metrics respectively. Detailed view in form of boxplots can be found in Appendix C.2.

Contrary to the original experiments from previous section, the difference in performance is much more pronounced. First, compare Tables 7.5 and 7.8 which show the aggregated EoR-R^2 results for the original experiments and experiments with simulated time-demanding evaluation respectively. While there was only a single case where an algorithm with fitness prediction was significantly better than the no-prediction algorithm in the original experiments, in the ones with simulated time-demanding evaluation there are multiple such cases, most of them for the WEC-A dataset. However, for the majority of the experiments with simulated time-demanding evaluation, the indifference of EoR-R^2 performance remained. On the other hand, this is not entirely unexpected as the goal of fitness prediction is to speed up the process rather than improve the end result.

Next, compare Tables 7.6 and 7.9 which show the aggregated RAUC performance for the original experiments and the ones with simulated time-demanding evaluation respectively. In the original experiments (Table 7.6), in almost all cases for GP and MGGP algorithms, no difference was identified, and for MGGP+LCF the fitness prediction algorithms were actually mostly worse than the base algorithm for the WEC-A dataset. On the other hand, in the experiments with simulated time-demanding evaluation (Table 7.6), although the negative trend for WEC-A dataset and MGGP+LCF algorithm remained in place, there is now a number of cases where an algorithm with fitness prediction was significantly better than the base algorithm.

The coevolution-based approach (*-CE-A and *-CE-E) turned out to be beneficial especially for the MGGP base algorithm as there was a significant improvement for majority of the experiments (4 datasets out of 6 for -A and 3 out of 6 for -E). The random-based approach (*-R-A and *-R-E) turned out to be beneficial especially for the GP base algorithm.

To complete the picture, the timing plots for experiments with simulated time-demanding evaluation are shown in Figures 7.9 through 7.14. As expected, the evaluation now takes the most time for all base algorithms on all datasets. Fitness prediction then decreases the proportion of evaluation in favour of the other parts of the algorithms.

Note on the number of iterations. With all the experiments, both normal and with simulated time-demanding evaluation, the algorithms were run for a fixed amount

[10] The time is shorter at the beginning of the algorithm and longer towards the end as the solutions tend to grow in size.

[11] For the MGGP+LCF algorithm, only the forward pass (see the end of Section 5.1.2) is delayed. Backpropagation of the error is unmodified as, technically, it is not the part of the evaluation.

	dataset	no pred.	*-CE-A	*-CE-E	*-LS-A	*-LS-E	*-R-A	*-R-E
GP-*	ASN	0.431	0.477	0.457	0.458	0.497	0.414	0.294
	ParkinsonMotor	0.200	0.388	0.334	0.255	0.343	0.391	0.299
	ParkinsonTotal	0.281	0.504	0.334	0.287	0.356	0.348	0.384
	puma8NH	0.672	0.670	0.674	0.667	0.666	0.675	0.673
	SupCon	0.434	**0.563**	0.471	0.510	**0.595**	**0.574**	**0.552**
	WEC-A	0.392	0.597	0.938	0.940	0.878	>0.999	>0.999
	total (+|=|-)	–	5|0|1	6|0|0	5|0|1	5|0|1	5|0|1	5|0|1
	total sig. (+|=|-)	–	1|5|0	0|6|0	0|6|0	1|5|0	1|5|0	1|5|0
MGGP-*	ASN	0.777	0.815	0.825	0.804	0.782	0.798	0.805
	ParkinsonMotor	0.670	0.736	0.714	0.712	0.702	0.720	0.730
	ParkinsonTotal	0.664	**0.739**	0.724	0.673	0.718	0.686	0.686
	puma8NH	0.681	0.679	0.681	0.682	0.684	0.683	0.683
	SupCon	0.689	**0.752**	**0.758**	0.747	0.741	0.725	0.731
	WEC-A	0.845	**>0.999**	**>0.999**	**0.999**	**0.962**	**0.937**	**0.945**
	total (+|=|-)	–	5|0|1	5|1|0	6|0|0	6|0|0	6|0|0	6|0|0
	total sig. (+|=|-)	–	3|3|0	2|4|0	1|5|0	1|5|0	1|5|0	1|5|0
MGGP+LCF-*	ASN	0.736	0.758	0.790	0.772	0.768	0.772	**0.778**
	ParkinsonMotor	0.401	0.442	0.353	0.510	0.449	0.467	0.408
	ParkinsonTotal	0.340	0.417	0.369	0.495	0.353	**0.549**	0.360
	puma8NH	0.672	0.672	0.673	0.671	0.671	0.672	0.674
	SupCon	0.712	0.734	0.722	0.725	0.730	0.722	0.721
	WEC-A	>0.999	**>0.999**	**>0.999**	**>0.999**	**>0.999**	**>0.999**	**>0.999**
	total (+|=|-)	–	5|1|0	6|0|0	5|0|1	5|0|1	5|1|0	6|0|0
	total sig. (+|=|-)	–	1|5|0	1|5|0	1|5|0	1|5|0	2|4|0	2|4|0

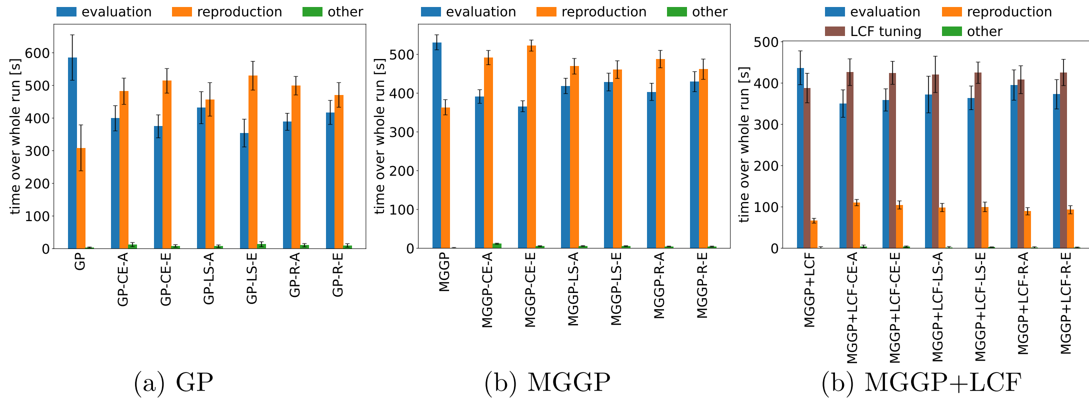
Table 7.8. Aggregated results of the fitness prediction algorithms on all datasets using the EoR R^2 performance characteristic for the experiments with simulated time-demanding evaluation. The "no pred." column shows the performance of the base algorithm without any prediction. The values are medians for the particular algorithm configurations and dataset over 20 independent runs. Values in green/red indicate that the median is better/worse than that of the base algorithm without prediction. Values set in bold indicate that their distributions also differ according to the two-sided MWUT with significance level $\alpha = 0.01$. The rows "total (+|=|-)" show the total number of times the particular configuration was better | equal to | worse than the base algorithm without prediction when comparing just the medians. The rows "total sig. (+|=|-)" are similar, but only the significant differences are considered better/worse.

of wall-clock time. Using a subsample of the training data for evaluation should cause the algorithm to perform more generations of the main evolution in total. We illustrate this in Table 7.10.

It can be seen that up to a few cases (which all happen for the vanilla GP-based algorithms), the algorithms with fitness prediction were able to perform more iterations than their no-prediction counterparts, i.e. they were able to test more candidate solutions. The difference is best seen on the larger datasets (they are sorted left to right from smallest to largest), and it is especially apparent with the simulated time-demanding evaluation (the bottom numbers in the table).

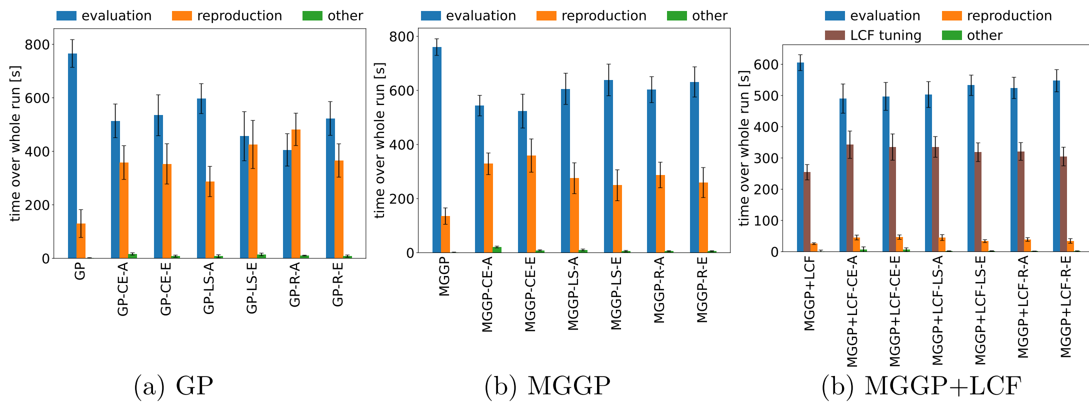
	dataset	no pred.	*-CE-A	*-CE-E	*-LS-A	*-LS-E	*-R-A	*-R-E
GP-*	ASN	0.270	0.291	0.300	0.352	0.296	0.332	0.165
	ParkinsonMotor	0.126	0.278	0.181	0.203	0.203	0.336	0.225
	ParkinsonTotal	0.145	**0.325**	0.239	0.220	0.182	0.292	**0.278**
	puma8NH	0.634	0.648	0.649	0.643	0.641	**0.659**	**0.663**
	SupCon	0.388	**0.530**	0.413	**0.440**	**0.511**	**0.532**	**0.503**
	WEC-A	0.121	0.558	0.431	0.346	0.644	**0.840**	**0.880**
	total (+|=|-)	–	6|0|0	6|0|0	6|0|0	6|0|0	6|0|0	5|0|1
	total sig. (+|=|-)	–	2|4|0	0|6|0	1|5|0	1|5|0	3|3|0	4|2|0
MGGP-*	ASN	0.743	0.765	0.782	0.726	0.748	0.769	0.776
	ParkinsonMotor	0.545	**0.627**	**0.638**	0.576	0.599	0.601	0.618
	ParkinsonTotal	0.589	**0.645**	**0.641**	0.607	0.603	0.611	0.610
	puma8NH	0.658	0.660	0.664	0.661	0.665	0.666	0.666
	SupCon	0.669	**0.711**	0.710	0.706	0.679	0.667	0.670
	WEC-A	0.691	**0.862**	**0.861**	**0.894**	**0.782**	**0.777**	**0.779**
	total (+|=|-)	–	5|0|1	6|0|0	5|0|1	6|0|0	5|0|1	6|0|0
	total sig. (+|=|-)	–	4|2|0	3|3|0	1|5|0	1|5|0	1|5|0	1|5|0
MGGP+LCF-*	ASN	0.696	0.717	0.725	0.696	0.721	0.724	0.728
	ParkinsonMotor	0.313	0.361	0.302	0.383	0.342	0.357	0.312
	ParkinsonTotal	0.288	0.344	0.301	0.369	0.296	**0.394**	0.289
	puma8NH	0.618	**0.642**	**0.636**	0.640	**0.639**	0.641	**0.639**
	SupCon	0.663	**0.678**	0.657	**0.707**	**0.678**	**0.703**	0.665
	WEC-A	0.949	**0.928**	**0.846**	**0.979**	**0.913**	**0.980**	**0.881**
	total (+|=|-)	–	5|0|1	3|0|3	5|1|0	5|0|1	6|0|0	4|0|2
	total sig. (+|=|-)	–	2|3|1	1|4|1	2|4|0	2|3|1	3|3|0	1|4|1

Table 7.9. Aggregated results of the fitness prediction algorithms on all datasets using the RAUC performance characteristic for the experiments with simulated time-demanding evaluation. The "no pred." column shows the performance of the base algorithm without any prediction. The values are medians for the particular algorithm configurations and dataset over 20 independent runs. Values in green/red indicate that the median is better/worse than that of the base algorithm without prediction. Values set in bold indicate that their distributions also differ according to the two-sided MWUT with significance level $\alpha = 0.01$. The rows "total (+|=|-)" show the total number of times the particular configuration was better | equal to | worse than the base algorithm without prediction when comparing just the medians. The rows "total sig. (+|=|-)" are similar, but only the significant differences are considered better/worse.



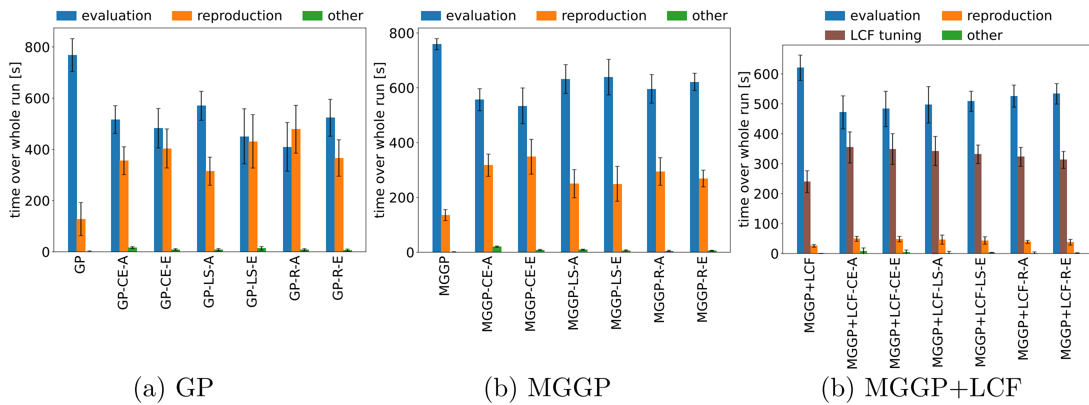
(a) GP (b) MGGP (b) MGGP+LCF

Figure 7.9. Timing of algorithms on the ASN dataset for the experiments with simulated time-demanding evaluation. The columns show mean time spent in the corresponding part of the algorithm, the error bars show standard deviation.



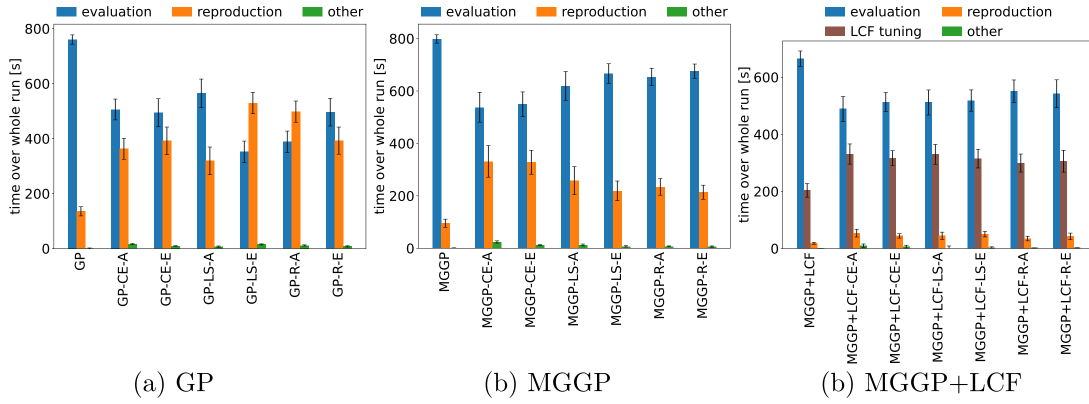
(a) GP (b) MGGP (b) MGGP+LCF

Figure 7.10. Timing of algorithms on the ParkinsonMotor dataset for the experiments with simulated time-demanding evaluation. The columns show mean time spent in the corresponding part of the algorithm, the error bars show standard deviation.



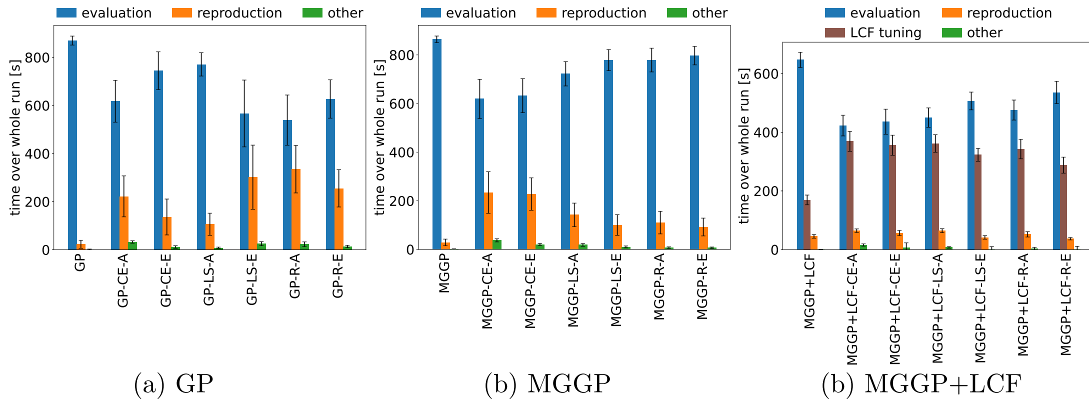
(a) GP (b) MGGP (b) MGGP+LCF

Figure 7.11. Timing of algorithms on the ParkinsonTotal dataset for the experiments with simulated time-demanding evaluation. The columns show mean time spent in the corresponding part of the algorithm, the error bars show standard deviation.



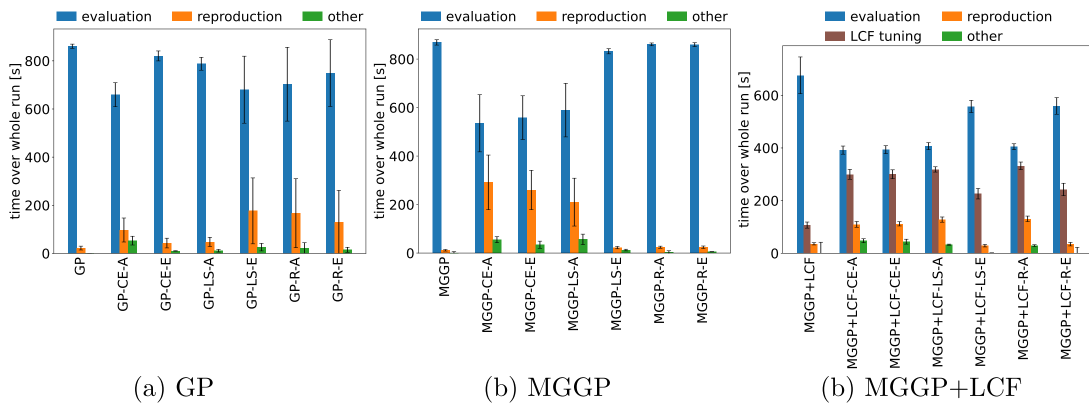
(a) GP (b) MGGP (b) MGGP+LCF

Figure 7.12. Timing of algorithms on the puma8NH dataset for the experiments with simulated time-demanding evaluation. The columns show mean time spent in the corresponding part of the algorithm, the error bars show standard deviation.



(a) GP (b) MGGP (b) MGGP+LCF

Figure 7.13. Timing of algorithms on the SupCon dataset for the experiments with simulated time-demanding evaluation. The columns show mean time spent in the corresponding part of the algorithm, the error bars show standard deviation.



(a) GP (b) MGGP (b) MGGP+LCF

Figure 7.14. Timing of algorithms on the WEC-A dataset for the experiments with simulated time-demanding evaluation. The columns show mean time spent in the corresponding part of the algorithm, the error bars show standard deviation.

algorithm	ASN	Parkinson Motor	Parkinson Total	puma8NH	SupCon	WEC-A
GP	783.3±645.6 384.1±109.0	387.7±192.7 183.5±46.4	726.9±818.0 197.5±49.6	364.0±83.3 135.2±9.3	509.4±332.0 81.2±4.5	184.7±56.0 23.6±0.7
GP-CE-A	1004.2±868.8 880.7±757.8	502.4±283.7 359.3±247.2	488.4±316.1 405.7±296.8	331.5±72.8 240.0±57.1	760.3±632.6 475.2±382.4	473.6±205.1 130.6±67.3
GP-CE-E	714.7±602.5 563.4±363.3	389.6±259.6 323.7±252.5	417.9±265.0 350.4±208.0	358.9±89.8 279.3±27.3	566.3±428.5 278.0±193.4	360.2±138.0 40.7±8.8
GP-LS-A	807.4±760.2 638.4±547.8	622.2±487.3 506.4±483.5	653.5±856.6 442.7±650.2	381.9±94.9 244.5±57.0	564.8±272.5 215.5±62.2	226.3±93.9 44.2±3.4
GP-LS-E	914.8±698.9 903.9±683.6	621.0±324.7 558.8±322.3	604.8±470.3 573.6±481.1	407.9±77.2 404.9±77.1	797.0±410.8 574.5±315.3	630.6±279.9 298.1±238.7
GP-R-A	1199.9±911.4 1131.9±910.8	502.8±219.4 480.4±191.0	431.3±259.2 407.4±264.4	377.3±45.5 366.6±64.3	865.0±541.2 674.7±455.1	638.1±233.2 293.1±266.4
GP-R-E	1140.6±924.7 1022.3±854.4	597.9±397.2 471.0±337.6	514.1±311.5 352.0±175.7	352.4±62.5 286.8±54.5	609.6±255.7 423.5±186.9	470.0±177.1 198.9±192.9
MGGP	300.2±31.3 233.9±17.7	249.4±45.1 138.8±15.2	237.6±47.7 141.5±17.2	387.7±100.8 142.1±10.1	367.8±142.2 75.2±2.5	143.2±31.6 23.0±0.4
MGGP-CE-A	322.0±41.8 298.2±33.6	324.6±79.9 235.2±50.7	324.7±80.9 226.1±48.2	402.3±87.6 296.4±69.3	388.6±100.1 262.8±70.9	604.2±173.2 379.5±129.2
MGGP-CE-E	308.0±81.7 296.9±39.6	339.2±75.3 262.1±46.1	321.6±66.4 243.9±83.5	436.3±126.3 268.5±94.1	418.8±120.4 272.9±57.5	640.4±134.5 346.6±84.3
MGGP-LS-A	328.0±39.8 280.8±35.9	315.3±102.5 229.1±67.2	315.7±67.3 215.0±50.2	419.6±98.5 267.0±46.1	465.8±118.8 231.9±89.8	572.6±105.2 304.4±124.8
MGGP-LS-E	320.3±48.4 295.6±44.8	323.0±55.8 227.9±48.5	329.4±81.7 233.7±66.1	436.9±79.3 262.8±49.9	424.6±150.9 201.7±56.6	270.7±77.6 44.9±2.3
MGGP-R-A	348.2±57.7 296.3±38.4	383.9±99.3 246.3±67.0	326.2±52.9 237.6±36.3	437.1±91.9 264.7±33.2	461.2±121.4 210.3±47.5	303.6±64.7 48.9±1.0
MGGP-R-E	351.2±57.4 321.1±41.7	328.0±79.4 241.4±57.3	334.4±65.9 237.5±40.2	440.7±87.1 258.2±41.3	425.1±96.8 207.0±49.7	381.1±124.3 48.5±1.0
MGGP+LCF	146.9±40.0 85.9±12.5	96.9±29.3 39.1±3.9	103.0±27.4 41.5±5.2	113.9±37.9 35.1±3.0	22.8±4.9 12.4±1.0	11.8±1.9 3.5±0.5
MGGP+LCF-CE-A	173.8±68.2 131.1±28.4	144.5±54.6 69.5±13.3	123.6±64.3 70.3±21.0	140.6±49.7 75.7±21.7	114.8±39.4 75.7±20.0	382.8±77.5 237.7±53.6
MGGP+LCF-CE-E	160.3±44.9 127.4±27.3	134.5±42.8 71.3±19.1	126.7±60.5 67.5±15.5	121.3±47.8 67.5±17.3	59.4±27.3 47.9±19.1	353.1±62.1 233.5±53.9
MGGP+LCF-LS-A	167.0±61.8 135.5±41.8	126.5±41.4 62.1±13.3	109.0±46.2 61.9±11.0	123.3±64.7 64.3±22.7	135.8±41.0 92.1±18.1	414.1±71.0 285.9±26.8
MGGP+LCF-LS-E	153.4±39.8 121.8±27.4	123.2±40.7 58.9±8.6	119.7±39.4 69.8±29.5	141.2±39.1 83.2±20.5	34.1±9.3 22.3±2.4	15.7±2.5 7.2±0.4
MGGP+LCF-R-A	184.3±52.1 126.0±28.8	106.8±34.5 57.1±8.9	114.8±28.0 57.8±8.1	133.0±54.5 57.1±13.1	65.8±52.2 56.0±51.2	408.7±84.4 285.1±48.4
MGGP+LCF-R-E	152.0±46.4 114.4±25.7	140.4±49.5 54.3±10.1	108.6±25.8 59.3±13.4	140.1±51.8 71.8±16.7	28.5±5.7 21.7±1.1	16.7±3.7 7.2±0.6

Table 7.10. Total number of generations performed by the algorithms (mean ± std. dev.). The top numbers in each cell show the value for the original experiments, the bottom numbers show the value for the experiments with simulated time-demanding evaluation.

7.6 Answers to research questions

With the experiments conducted, we return to the research question we laid out in Section 7.3 to answer them.

RQ1: Can uniform random sampling replace the inner GA as the algorithm for finding suitable fitness predictors, while keeping or improving the performance? For the normal experiments, none of the fitness prediction approaches were, overall, able to significantly improve the performance of the base algorithm, with the exception of MGGP+LCF algorithm in the RAUC metric (see Table 7.6).

In the experiments with simulated time-demanding evaluation, random sampling turned out to be surprisingly effective for GP (from the point of view of the RAUC metric, see Table 7.9). For MGGP, random sampling turned out to be only very slightly more effective than not using any fitness prediction, while coevolution provided greater improvement. For MGGP+LCF, one of the two random sampling configurations proved to be somewhat effective both in normal experiments, where it brought slightly better improvement than coevolution, and in simulated time-demanding experiments, where the performance was comparable to that of coevolution.

There is a possible analogy to mini-batch stochastic gradient descent (SGD) where the mini-batches are also randomly sampled. Randomly sampled subset provides an unbiased view on the data which cannot be said about neither of the other fitness prediction methods. Another supporting evidence is reported in [101], where they examined possibilities of biasing the selection of fitness cases in lexicase selection [102]. They found out that the biased shufflings did not improve the performance compared to uniform random shuffling.

From our point of view, it can be stated that random sampling is not inferior and should be considered as a valid option, especially when parameter tuning is an issue as there are no parameters associated with the subset selection.

RQ2: Can local search replace the inner GA as the algorithm for finding suitable fitness predictors, while keeping or improving the performance? In the normal experiments, the performance of local search was very similar to that of coevolution, as neither approach was able to improve the performance of the base algorithm very much (see Tables 7.5 and 7.6).

In the experiments with simulated time-demanding evaluation, local search seems to perform similarly to coevolution for GP, slightly better than coevolution for MGGP+LCF, and worse than coevolution for MGGP. It can therefore be said that unless MGGP algorithm is used, local search can replace the inner GA without a negative impact on performance while reducing the number of parameters and simplifying the algorithm.

RQ3: Are fitness predictors beneficial when LCFs (see Chapter 5) are used? Based on the results, we cannot provide a universal answer to this question. It seems that when LCFs alone are enough to explain the data (as the WEC-A dataset seems to be), the answer tends towards no, with the exception of the MGGP+LCF-R-A configuration which has shown improvement over the baseline both in normal experiments and experiments with simulated time-demanding evaluation (in terms of RAUC). Regarding other datasets, there is a net positive effect, but its magnitude is questionable, especially considering the need to determine the parameters of the fitness prediction method.

7.7 Summary and conclusion

In this chapter, we analysed the topic of case-based fitness prediction in SR. The main goal of fitness prediction is to make the algorithm evolve the solutions faster by reducing the amount of fitness computation through using a subset of training datapoints.

We proposed two modifications of the approach from [94] by replacing the inner GA with a simple uniform random sampling, and with a simple greedy local search procedure. The potential benefit of using a simpler search algorithm is fewer parameters to choose and easier implementation, if needed. We also tested how the fitness prediction approaches compare on larger and more complex datasets than in previous research.

The results have shown that fitness prediction, in general, is not an universally applicable technique. In the basic experiments, we found little to no improvement over the base algorithms without fitness prediction. However, when we artificially emphasized the time spent in evaluation, fitness prediction became a significant improvement. This indicates that implementation and the design of the base algorithm are also important when considering the usage of fitness prediction. If the algorithm is fast compared to evaluation, fitness prediction seems to be beneficial, while if it is the other way around, the benefit is smaller or even none. This also suggests that fitness prediction might be useful for other, non-SR applications of GP where the evaluation is more computationally demanding.

Our proposal of replacing the inner GA with random sampling turned out to be effective (better than the coevolution approach) when the base algorithm is vanilla GP (when evaluation time is emphasized). For MGGP+LCF, random sampling turned out to be more effective than coevolution but not by such a margin as for vanilla GP. Random sampling was worse for MGGP. Our modification of replacing the inner GA with local search turned out to be, overall, about as effective as coevolution, except for the MGGP algorithm, where coevolution was the best approach.

Based on these results, we consider random sampling and local search as a competitive alternatives to coevolution, while being simpler to implement and requiring lower number of parameters to be set.

Chapter 8
Conclusion

In this thesis, we focused on the task of symbolic regression by genetic programming-based algorithms. We first examined the state of the art of symbolic regression where we have shown that nowadays a common technique is to form and evolve (in algorithms which use evolution) models in the form of linear combination of multiple (non-linear) basis functions, utilizing some kind of linear regression to estimate the coefficients of the combination. We have selected four of the state-of-the-art algorithms and performed an empirical comparison on a common set of benchmarks which has not been done so far with these algorithms. We have found out that none of the tested algorithms was universally superior to the others. We have also found out that the SR algorithms are, generally, not better than conventional machine learning techniques like random forests or support vector regression machines, but they still produce a symbolic model which the conventional algorithms do not. Based on the results we selected the MGGP algorithm for further extensions, although those are applicable to any GPbased algorithm.

To (partially) mitigate both the problems of simultaneous learning of structure and parameters, and the problem of blindness of search, we proposed an extension to GP-based algorithms in the form of new type of node, the Linear Combination of Features. The LCF nodes encapsulate all the problem features in a single linear combination with coefficients tuned in an informed way by utilizing the error gradient. The LCFs can easily encode affine transformations of the feature space which we have demonstrated with simple proof-of-concept experiments. We have subjected several variants of this extension to testing on both artificial and real-world datasets and we found it overall beneficial to the performance of the algorithm when gradient-based tuning of the parameters is used.

We have also introduced the task of dynamic symbolic regression with small gradual changes which stems from the value iteration algorithm from the field of reinforcement learning. We proposed to use the new leaf node for this task as we argued that small changes can be well captured by adjusting the parameters of the linear combinations. Both proof-of-concept experiment and testing on RL-based problems has shown that LCFs indeed help the algorithm to better track the changing target.

Finally, we have examined the technique of fitness prediction based on subsampling the training dataset. The main idea of this kind of fitness prediction is to save time by not evaluating the candidate solutions on all datapoints but only those that sufficiently represent the dataset. We based upon previous work with coevolved predictors, that introduced adaptation of the size of the subset. We proposed two simplifications of the algorithm that lie in replacing the coevolution with random sampling and local search. The first set of experiments did show only little difference between using fitness predictors and full training set. The most benefit was brought by one of the random sampling configurations on the MGGP algorithm with LCFs. We hypothesized that the cause might be the relatively low-cost evaluation compared to the rest of the algorithm. Therefore, we conduced a second set of experiments where we simulated a more time-demanding evaluation by introducing an artificial delay to the evaluation process. These

experiments have shown a clear advantage of the algorithms with fitness prediction. The coevolution-based approach was most effective for the MGGP algorithm without LCFs, while the random sampling approach was most effective on vanilla GP and MGGP with LCFs. The local search approach was the most effective on MGGP+LCF as well, but not very much on the other algorithms. We conclude that coevolution is not a necessary component and can be replaced with a simpler algorithm (that requires less parameters or even none) in some cases.

8.1 Future work

Expand benchmarking study. The comparison presented in Chapter 4 provides a basic insight into the performance differences between the selected methods and further benchmarking might reveal patterns not discovered there. The most straightforward way is to expand the set of algorithms, notably the recent ones we discussed in Chapter 3, and benchmarks (with varying complexities, higher number of dimensions, and noise), providing a broader and more robust comparison of current state of the art.

Another view on the algorithm comparison may be provided by unifying the sets of function symbols of all compared algorithms (which would, however, require generalizations and non-trivial changes to some of the presented algorithms' implementations) and possibly other settings as well. The expanded set of benchmarks shall also allow to tune the available parameters of the methods, and thus reduce the effects caused by possibly suboptimal parameter settings.

Different change dynamic in dynamic SR. In Chapter 6 we examined dynamic behaviour in a setting where the target changes at regular time intervals of the length which was one to four minutes long. It would certainly be interesting to see how would the algorithms cope with this time getting shorter, up to the extreme of switching to the next stage every generation. We can reasonably argue that restarting would get only worse results but the performance of LCFs is unknown, yet worth exploring.

Effectiveness of fitness predictors in context of algorithm design and implementation. As we have shown in Chapter 7, the effectiveness of fitness prediction greatly increased when we introduced an artificial delay to the fitness evaluation. Disregarding evaluation, different algorithms have different computational demands and therefore the amount of time saved by using fitness prediction can vary. Investigation in this effect could provide valuable insight as well as similar investigation regarding the actual implementation.

References

[1] Holland, John H. *Adaptation in Natural and Artificial Systems.* Cambridge, MA, USA: MIT Press, 1992. ISBN 0-262-58111-6.

[2] Brindle, Anne. *Genetic algorithms for function optimization.* Alberta: University of Alberta, 1980. Ph.D. Thesis.

[3] Goldberg, David E., and Kalyanmoy Deb. *A Comparative Analysis of Selection Schemes Used in Genetic Algorithms.* Available from DOI 10.1016/B978-0-08-050684-5.50008-2.

[4] Koza, John R. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* Cambridge, MA, USA: MIT Press, 1992. ISBN 0-262-11170-5. Available from `http://mitpress.mit.edu/books/genetic-programming`.

[5] Evett, Matthew P., and Thomas Fernandez. Numeric Mutation: Improved Search in Genetic Programming. In: *Proceedings of the Eleventh International Florida Artificial Intelligence Research Society Conference.* AAAI Press, 1998. pp. 106–109. ISBN 1577350510.

[6] Schmidt, Michael, and Hod Lipson. Distilling free-form natural laws from experimental data. *Science.* American Association for the Advancement of Science, 2009, Vol. 324, No. 5923, pp. 81–85.

[7] Schmidt, Michael, and Hod Lipson. *Eureqa (Version 0.98 beta) [Software].* Available from `www.nutonian.com`.

[8] Nelder, J. A., and R. W. M. Wedderburn. Generalized Linear Models. *Journal of the Royal Statistical Society. Series A (General).* Royal Statistical Society, Wiley, 1972, Vol. 135, No. 3, pp. 370–384. ISSN 00359238. Available from `http://www.jstor.org/stable/2344614`. Accessed: 2021-07-17.

[9] Keijzer, Maarten. Scaled Symbolic Regression. *Genetic Programming and Evolvable Machines.* 2004, Vol. 5, No. 3, pp. 259–269. ISSN 1573-7632. Available from DOI 10.1023/B:GENP.0000030195.77571.f9.

[10] Hinchliffe, Mark et al. Modelling Chemical Process Systems Using a Multi-Gene Genetic Programming Algorithm. In: *Late Breaking Paper, GP'96.* Stanford, USA, 1996. pp. 56–65.

[11] Searson, Dominic, Mark Willis, and Gary Montague. Co-evolution of non-linear PLS model components. *Journal of Chemometrics.* 2007, Vol. 21, No. 12, pp. 592–603. Available from DOI https://doi.org/10.1002/cem.1084.

[12] Garg, Akhil, Ankit Garg, and K. Tai. A multi-gene genetic programming model for estimating stress-dependent soil water retention curves. *Computational Geosciences.* 2013, Vol. 18, No. 1, pp. 45–56. ISSN 1573-1499. Available from DOI 10.1007/s10596-013-9381-z.

[13] Arnaldo, Ignacio, Krzysztof Krawiec, and Una-May O'Reilly. Multiple Regression Genetic Programming. In: *Proceedings of the 2014 Annual Conference*

on Genetic and Evolutionary Computation. New York, NY, USA: ACM, 2014. pp. 879–886. GECCO '14. ISBN 978-1-4503-2662-9. Available from DOI 10.1145/2576768.2598291.

[14] DEB, Kalyanmoy et al. A Fast Elitist Non-dominated Sorting Genetic Algorithm for Multi-objective Optimization: NSGA-II. In: Marc SCHOENAUER, Kalyanmoy DEB, Günther RUDOLPH et al., eds. *Parallel Problem Solving from Nature PPSN VI*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000. pp. 849–858. ISBN 978-3-540-45356-7. Available from DOI 10.1007/3-540-45356-3_83.

[15] ARNALDO, Ignacio, Una-May O'REILLY, and Kalyan VEERAMACHANENI. Building Predictive Models via Feature Synthesis. In: *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*. New York, NY, USA: ACM, 2015. pp. 983–990. GECCO '15. ISBN 978-1-4503-3472-3. Available from DOI 10.1145/2739480.2754693.

[16] MCCONAGHY, Trent. *FFX: Fast, Scalable, Deterministic Symbolic Regression Technology*. Available from DOI 10.1007/978-1-4614-1770-5_13.

[17] FRIEDMAN, Jerome, Trevor HASTIE, and Rob TIBSHIRANI. Regularization Paths for Generalized Linear Models via Coordinate Descent. *Journal of Statistical Software*. 2010, Vol. 33, No. 1, pp. 1–22. ISSN 1548-7660. Available from DOI 10.18637/jss.v033.i01.

[18] ZOU, Hui, and Trevor HASTIE. Regularization and variable selection via the elastic net. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*. Blackwell Publishing Ltd, 2005, Vol. 67, No. 2, pp. 301–320. ISSN 1467-9868. Available from DOI 10.1111/j.1467-9868.2005.00503.x.

[19] MORAGLIO, Alberto, Krzysztof KRAWIEC, and ColinG. JOHNSON. *Geometric Semantic Genetic Programming*. Available from DOI 10.1007/978-3-642-32937-1_3.

[20] MARTINS, Joao Francisco B. S. et al. Solving the Exponential Growth of Symbolic Regression Trees in Geometric Semantic Genetic Programming. In: *Proceedings of the Genetic and Evolutionary Computation Conference*. New York, NY, USA: ACM, 2018. pp. 1151–1158. GECCO '18. ISBN 978-1-4503-5618-3. Available from DOI 10.1145/3205455.3205593.

[21] SCHMIDT, M. D., and H. LIPSON. Coevolution of Fitness Predictors. *IEEE Transactions on Evolutionary Computation*. 2008, Vol. 12, No. 6, pp. 736-749. Available from DOI 10.1109/TEVC.2008.919006.

[22] SCHMIDT, Michael D., and Hod LIPSON. Age–Fitness Pareto Optimization. In: *Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation*. New York, NY, USA: Association for Computing Machinery, 2010. pp. 543–544. GECCO '10. ISBN 9781450300728. Available from DOI 10.1145/1830483.1830584.

[23] SMITS, Guido F., and Mark KOTANCHEK. Pareto-Front Exploitation in Symbolic Regression. In: Una-May O'REILLY et al., eds. *Genetic Programming Theory and Practice II*. Boston, MA: Springer US, 2005. pp. 283–299. ISBN 978-0-387-23254-6. Available from DOI 10.1007/0-387-23254-0_17.

[24] VLADISLAVLEVA, Ekaterina, Guido SMITS, and Mark KOTANCHEK. Better Solutions Faster: Soft Evolution of Robust Regression Models In Pareto Genetic Programming. In: Rick RIOLO, Terence SOULE, and Bill WORZEL, eds. *Genetic Programming Theory and Practice V*. Boston, MA: Springer US, 2008. pp. 13–32. ISBN 978-0-387-76308-8. Available from DOI 10.1007/978-0-387-76308-8_2.

[25] Worm, Tony, and Kenneth Chiu. Prioritized Grammar Enumeration: Symbolic Regression by Dynamic Programming. In: *Proceedings of the 15th Annual Conference on Genetic and Evolutionary Computation*. New York, NY, USA: ACM, 2013. pp. 1021–1028. GECCO '13. ISBN 978-1-4503-1963-8. Available from DOI 10.1145/2463372.2463486.

[26] De Melo, Vinicius Veloso. Kaizen Programming. In: *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation*. New York, NY, USA: ACM, 2014. pp. 895–902. GECCO '14. ISBN 978-1-4503-2662-9. Available from DOI 10.1145/2576768.2598264.

[27] Cava, William La et al. Learning concise representations for regression by evolving networks of trees. In: *International Conference on Learning Representations*. 2019. Available from `https://openreview.net/forum?id=Hke-JhA9Y7`. Accessed: 2021-05-25.

[28] La Cava, William, Lee Spector, and Kourosh Danai. Epsilon-Lexicase Selection for Regression. In: *Proceedings of the Genetic and Evolutionary Computation Conference 2016*. New York, NY, USA: Association for Computing Machinery, 2016. pp. 741–748. GECCO '16. ISBN 9781450342063. Available from DOI 10.1145/2908812.2908898.

[29] La Cava, William, and Jason H. Moore. Semantic Variation Operators for Multidimensional Genetic Programming. In: *Proceedings of the Genetic and Evolutionary Computation Conference*. New York, NY, USA: Association for Computing Machinery, 2019. pp. 1056–1064. GECCO '19. ISBN 9781450361118. Available from DOI 10.1145/3321707.3321776.

[30] Udrescu, Silviu-Marian, and Max Tegmark. AI Feynman: A physics-inspired method for symbolic regression. *Science Advances*. American Association for the Advancement of Science, 2020, Vol. 6, No. 16. Available from DOI 10.1126/sciadv.aay2631.

[31] Petersen, Brenden K. et al. Deep symbolic regression: Recovering mathematical expressions from data via risk-seeking policy gradients. In: *International Conference on Learning Representations*. 2021. Available from `https://openreview.net/forum?id=m5Qsh0kBQG`. Accessed: 2021-06-24.

[32] *EFS commit* `6d991fa`. Available from `http://github.com/flexgp/efs/tree/6d991fa`.

[33] Searson, Dominic P. GPTIPS 2: An Open-Source Software Platform for Symbolic Data Mining. In: H. Amir Gandomi, H. Amir Alavi, and Conor Ryan, eds. *Handbook of Genetic Programming Applications*. Cham: Springer International Publishing, 2015. pp. 551–573. ISBN 978-3-319-20883-1. Available from DOI 10.1007/978-3-319-20883-1_22.

[34] McDermott, James et al. Genetic Programming Needs Better Benchmarks. In: *Proceedings of the 14th Annual Conference on Genetic and Evolutionary Computation*. New York, NY, USA: ACM, 2012. pp. 791–798. GECCO '12. ISBN 978-1-4503-1177-9. Available from DOI 10.1145/2330163.2330273.

[35] Korns, Michael F. Genetic Programming Theory and Practice IX. In: Rick Riolo, Ekaterina Vladislavleva, and H. Jason Moore, eds. New York, NY: Springer New York, 2011. pp. 129–151. ISBN 978-1-4614-1770-5. Available from DOI 10.1007/978-1-4614-1770-5_8.

[36] Vladislavleva, E. J., G. F. Smits, and D. den Hertog. Order of Nonlinearity as a Complexity Measure for Models Generated by Symbolic Regression via Pareto Genetic Programming. *Evolutionary Computation, IEEE Transactions on.* April, 2009, Vol. 13, No. 2, pp. 333–349. ISSN 1089-778X. Available from DOI 10.1109/TEVC.2008.926486.

[37] Tsanas, Athanasios, and Angeliki Xifara. Accurate quantitative estimation of energy performance of residential buildings using statistical machine learning tools. *Energy and Buildings.* 2012, Vol. 49, pp. 560–567. ISSN 0378-7788. Available from DOI 10.1016/j.enbuild.2012.03.003.

[38] Bache, K., and M. Lichman. *UCI Machine Learning Repository*. Available from `http://archive.ics.uci.edu/ml`. Accessed: 2021-05-18.

[39] Yeh, I.-C.. Modeling of strength of high-performance concrete using artificial neural networks. *Cement and Concrete Research.* 1998, Vol. 28, No. 12, pp. 1797–1808. ISSN 0008-8846. Available from DOI 10.1016/S0008-8846(98)00165-3.

[40] Pedregosa, F. et al. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research.* 2011, Vol. 12, pp. 2825–2830.

[41] *scikit-learn 0.17.1.* Available from `https://pypi.python.org/pypi/scikit-learn/0.17.1`.

[42] Ni, J., R. H. Drieberg, and P. I. Rockett. The Use of an Analytic Quotient Operator in Genetic Programming. *IEEE Transactions on Evolutionary Computation.* Feb, 2013, Vol. 17, No. 1, pp. 146-152. ISSN 1089-778X. Available from DOI 10.1109/TEVC.2012.2195319.

[43] Searson, Dominic P.. *GPTIPS 2.* Available from `http://sites.google.com/site/gptips4matlab`. Accessed: 2017-03-09.

[44] *FFX 1.3.4.* Available from `http://pypi.python.org/pypi/ffx/1.3.4`.

[45] *GSGP-Red commit* `0e5f4d5`. Available from `https://github.com/laic-ufmg/GSGP-Red/tree/0e5f4d5`.

[46] Mann, H. B., and D. R. Whitney. On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other. *The Annals of Mathematical Statistics.* Institute of Mathematical Statistics, 1947, Vol. 18, No. 1, pp. 50–60. ISSN 00034851. Available from DOI 10.1214/aoms/1177730491.

[47] Wilcoxon, Frank. Individual Comparisons by Ranking Methods. *Biometrics Bulletin.* [International Biometric Society, Wiley, 1945, Vol. 1, No. 6, pp. 80–83. ISSN 00994987. Available from DOI 10.2307/3001968.

[48] Searson, Dominic P., David E. Leahy, and Mark J. Willis. GPTIPS: an open source genetic programming toolbox for multigene symbolic regression. In: *Proceedings of the International MultiConference of Engineers and Computer Scientists.* 2010. pp. 77–80.

[49] Olson, Randal S. et al. PMLB: a large benchmark suite for machine learning evaluation and comparison. *BioData Mining.* Dec, 2017, Vol. 10, No. 1, pp. 36. ISSN 1756-0381. Available from DOI 10.1186/s13040-017-0154-4.

[50] Kubalík, Jiří, Erik Derner, and Robert Babuška. Enhanced symbolic regression through local variable transformations. In: *Proceedings of the 9th International Joint Conference on Computational Intelligence (IJCCI 2017).* SciTePress, 2017. pp. 91–100. ISBN 978-989-758-274-5. Available from DOI 10.5220/0006505200910100.

[51] Rumelhart, David E., Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*. Oct, 1986, Vol. 323, No. 6088, pp. 533–536. Available from DOI 10.1038/323533a0.

[52] Topchy, Alexander, and William F Punch. Faster genetic programming based on local gradient search of numeric leaf values. In: *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*. 2001. Available from DOI 10.5555/2955239.2955258.

[53] Zhang, Mengjie, and Will Smart. Genetic Programming with Gradient Descent Search for Multiclass Object Classification. In: Maarten Keijzer et al., eds. *Genetic Programming*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004. pp. 399–408. ISBN 978-3-540-24650-3. Available from DOI 10.1007/978-3-540-24650-3_38.

[54] Kommenda, Michael et al. Effects of Constant Optimization by Nonlinear Least Squares Minimization in Symbolic Regression. In: *Proceedings of the 15th Annual Conference Companion on Genetic and Evolutionary Computation*. New York, NY, USA: ACM, 2013. pp. 1121–1128. GECCO '13 Companion. ISBN 978-1-4503-1964-5. Available from DOI 10.1145/2464576.2482691.

[55] Smart, Will, and Mengjie Zhang. Continuously evolving programs in genetic programming using gradient descent. In: *Proceedings of the 7th Asia-Pacific Conference on Complex Systems Cairns Convention Centre, Cairns, Australia, 6-10th December 2004*. 2004.

[56] Igel, Christian, and Michael Hüsken. Improving the Rprop learning algorithm. In: *Proceedings of the second international ICSC symposium on neural computation (NC 2000)*. 2000. pp. 115–121.

[57] Igel, Christian, and Michael Hüsken. Empirical evaluation of the improved Rprop learning algorithms. *Neurocomputing*. 2003, Vol. 50, pp. 105–123. ISSN 0925-2312. Available from DOI 10.1016/S0925-2312(01)00700-7.

[58] Jackson, David. Genetic Programming: 15th European Conference, EuroGP 2012, Málaga, Spain, April 11–13, 2012. Proceedings. In: Alberto Moraglio et al., eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012. pp. 49–60. ISBN 978-3-642-29139-5. Available from DOI 10.1007/978-3-642-29139-5_5.

[59] Jackson, David. Single Node Genetic Programming on Problems with Side Effects. In: Carlos A. Coello Coello, Vincenzo Cutello, Kalyanmoy Deb, Stephanie Forrest, Giuseppe Nicosia, and Mario Pavone, eds. *Parallel Problem Solving from Nature - PPSN XII*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012. pp. 327–336. ISBN 978-3-642-32937-1.

[60] Žegklitz, Jan. *Evo*. Available from `https://github.com/zegkljan/evo`. Accessed: 2018-11-27.

[61] Standard Performance Evaluation Corporation. *Floating Point Component of SPEC CPU2006*. Available from `https://www.spec.org/cpu2006/CFP2006`.

[62] Alibekov, Eduard, Jiří Kubalík, and Robert Babuška. Symbolic method for deriving policy in reinforcement learning. In: *2016 IEEE 55th Conference on Decision and Control (CDC)*. 2016. pp. 2789–2795. Available from DOI 10.1109/CDC.2016.7798684.

[63] Alibekov, Eduard, Jiří Kubalík, and Robert Babuška. Policy derivation methods for critic-only reinforcement learning in continuous spaces. *Engineering Ap-*

plications of Artificial Intelligence. 2018, Vol. 69, pp. 178–187. ISSN 0952-1976. Available from DOI 10.1016/j.engappai.2017.12.004.

[64] Nguyen, Trung Thanh, Shengxiang Yang, and Juergen Branke. Evolutionary dynamic optimization: A survey of the state of the art. *Swarm and Evolutionary Computation*. 2012, Vol. 6, No. Supplement C, pp. 1–24. ISSN 2210-6502. Available from DOI 10.1016/j.swevo.2012.05.001.

[65] Goldberg, David E., and Robert E. Smith. Nonstationary Function Optimization Using Genetic Algorithm with Dominance and Diploidy. In: *Proceedings of the Second International Conference on Genetic Algorithms on Genetic Algorithms and Their Application*. Hillsdale, NJ, USA: L. Erlbaum Associates Inc., 1987. pp. 59–68. ISBN 0-8058-0158-8.

[66] Yang, Shengxiang. On the Design of Diploid Genetic Algorithms for Problem Optimization in Dynamic Environments. In: *2006 IEEE International Conference on Evolutionary Computation*. 2006. pp. 1362-1369. ISSN 1089-778X. Available from DOI 10.1109/CEC.2006.1688467.

[67] Branke, J. Memory enhanced evolutionary algorithms for changing optimization problems. In: *Proceedings of the 1999 Congress on Evolutionary Computation-CEC99 (Cat. No. 99TH8406)*. 1999. pp. 1882 Vol. 3. Available from DOI 10.1109/CEC.1999.785502.

[68] Yang, S., and X. Yao. Population-Based Incremental Learning With Associative Memory for Dynamic Environments. *IEEE Transactions on Evolutionary Computation*. Oct, 2008, Vol. 12, No. 5, pp. 542-561. ISSN 1089-778X. Available from DOI 10.1109/TEVC.2007.913070.

[69] Simões, Anabela, and Ernesto Costa. Memory-based CHC Algorithms for the Dynamic Traveling Salesman Problem. In: *Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation*. New York, NY, USA: ACM, 2011. pp. 1037–1044. GECCO '11. ISBN 978-1-4503-0557-0. Available from DOI 10.1145/2001576.2001717.

[70] Eggermont, Jeroen et al. Raising the dead: Extending evolutionary algorithms with a case-based memory. *Genetic Programming*. Springer, 2001, pp. 280–290. Available from DOI 10.1007/3-540-45355-5_22.

[71] Cobb, Helen G. *An investigation into the use of hypermutation as an adaptive operator in genetic algorithms having continuous, time-dependent nonstationary environments*. Technical Report AIC-90-001, Naval Research Laboratory, Washington, DC. Available from `https://apps.dtic.mil/sti/citations/ADA229159`. Accessed: 2019-06-12.

[72] Vavak, Frank, Ken Jukes, and Terence C Fogarty. Learning the local search range for genetic optimisation in nonstationary environments. In: *Evolutionary Computation, 1997., IEEE International Conference on*. 1997. pp. 355–360. Available from DOI 10.1109/ICEC.1997.592335.

[73] Vavak, Frank et al. Performance of a genetic algorithm with variable local search range relative to frequency of the environmental changes. *Genetic Programming*. 1998, pp. 22–25.

[74] Grefenstette, John J. Genetic algorithms for changing environments. In: *PPSN*. 1992. pp. 137–144.

[75] ANDERSEN, HC. *An investigation into genetic algorithms, and the relationship between speciation and the tracking of optima in dynamic functions*. Brisbane, Australia: Queensland University of Technology, 1991. Honours thesis.

[76] *Handbook of Evolutionary Computation*. 1st ed. Bristol, UK, UK: Bristol: Adam Hilger, IOP Publishing, 1997. ISBN 0750303921. Available from DOI 10.1201/9780367802486.

[77] BUI, L. T., H. A. ABBASS, and J. BRANKE. Multiobjective optimization for dynamic environments. In: *2005 IEEE Congress on Evolutionary Computation*. 2005. pp. 2349-2356 Vol. 3. ISSN 1089-778X. Available from DOI 10.1109/CEC.2005.1554987.

[78] TUITE, Cliodhna, Michael O'NEILL, and Anthony BRABAZON. Towards a Dynamic Benchmark for Genetic Programming. In: *Proceedings of the 15th Annual Conference Companion on Genetic and Evolutionary Computation*. New York, NY, USA: ACM, 2013. pp. 151–152. GECCO '13 Companion. ISBN 978-1-4503-1964-5. Available from DOI 10.1145/2464576.2464649.

[79] MACEDO, João, Ernesto COSTA, and Lino MARQUES. Genetic Programming Algorithms for Dynamic Environments. In: Giovanni SQUILLERO, and Paolo BURELLI, eds. *Applications of Evolutionary Computation*. Cham: Springer International Publishing, 2016. pp. 280–295. ISBN 978-3-319-31153-1.

[80] YANG, Shengxiang, and Renato TINÓS. A hybrid immigrants scheme for genetic algorithms in dynamic environments. *International Journal of Automation and Computing*. Jul, 2007, Vol. 4, No. 3, pp. 243–254. ISSN 1751-8520. Available from DOI 10.1007/s11633-007-0243-9.

[81] SIMÕES, Anabela, and Ernesto COSTA. On Biologically Inspired Genetic Operators: Transformation in the Standard Genetic Algorithm. In: *Proceedings of the 3rd Annual Conference on Genetic and Evolutionary Computation*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001. pp. 584–591. GECCO'01. ISBN 1-55860-774-9. Available from `https://dl.acm.org/doi/10.5555/2955239.2955328`. Accessed: 2019-06-12.

[82] GALVÁN-LÓPEZ, Edgar et al. On the Use of Dynamic GP Fitness Cases in Static and Dynamic Optimisation Problems. In: *EA 2017-International Conference on Artificial Evolution*. 2017. pp. 80–93. ISBN 978-2-9539267-7-4. Available from DOI 10.1007/978-3-319-78133-4_6.

[83] SLANY, K.. Cartesian Genetic Programming in a changing environment. In: *2015 7th International Joint Conference on Computational Intelligence (IJCCI)*. 2015. pp. 204–211.

[84] SUTTON, Richard S., and Andrew G BARTO. *Reinforcement Learning: An Introduction*. Cambridge, Massachusetts: MIT Press, 2020. ISBN 9780262039246.

[85] TROJANOWSKI, K., and Z. MICHALEWICZ. Searching for optima in non-stationary environments. In: *Proceedings of the 1999 Congress on Evolutionary Computation-CEC99 (Cat. No. 99TH8406)*. 1999. pp. 1850 Vol. 3. Available from DOI 10.1109/CEC.1999.785498.

[86] GATHERCOLE, Chris, and Peter ROSS. Dynamic training subset selection for supervised learning in Genetic Programming. In: Yuval DAVIDOR, Hans-Paul SCHWEFEL, and Reinhard MÄNNER, eds. *Parallel Problem Solving from Nature — PPSN III*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1994. pp. 312–321. ISBN 978-3-540-49001-2. Available from DOI 10.1007/3-540-58484-6_275.

[87] Liu, Yi, and T. Khoshgoftaar. Reducing overfitting in genetic programming models for software quality classification. In: *High Assurance Systems Engineering, 2004. Proceedings. Eighth IEEE International Symposium on.* 2004. pp. 56-65. ISSN 1530-2059. Available from DOI 10.1109/HASE.2004.1281730.

[88] Gonçalves, Ivo, and Sara Silva. Experiments on controlling overfitting in genetic programming. In: *15th Portuguese conference on artificial intelligence (EPIA 2011).* 2011. ISBN 978-989-95618-4-7.

[89] Gonçalves, Ivo et al. Random Sampling Technique for Overfitting Control in Genetic Programming. In: Alberto Moraglio et al., eds. *Genetic Programming*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012. pp. 218–229. ISBN 978-3-642-29139-5. Available from DOI 10.1007/978-3-642-29139-5_19.

[90] Gonçalves, Ivo, and Sara Silva. *Balancing Learning and Overfitting in Genetic Programming with Interleaved Sampling of Training Data*. Available from DOI 10.1007/978-3-642-37207-0_7.

[91] Šikulová, Michaela, and Lukáš Sekanina. Coevolution in Cartesian Genetic Programming. In: Alberto Moraglio et al., eds. *Genetic Programming*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012. pp. 182–193. ISBN 978-3-642-29139-5.

[92] Schmidt, Michael D., and Hod Lipson. Predicting Solution Rank to Improve Performance. In: *Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation*. New York, NY, USA: Association for Computing Machinery, 2010. pp. 949–956. GECCO '10. ISBN 9781450300728. Available from DOI 10.1145/1830483.1830652.

[93] Ly, Daniel L., and Hod Lipson. Trainer selection strategies for coevolving rank predictors. In: *2011 IEEE Congress of Evolutionary Computation (CEC).* 2011. pp. 2399–2406. Available from DOI 10.1109/CEC.2011.5949914.

[94] Drahosova, M., L. Sekanina, and M. Wiglasz. Adaptive Fitness Predictors in Coevolutionary Cartesian Genetic Programming. *Evolutionary Computation*. 2019, Vol. 27, No. 3, pp. 497-523. Available from DOI 10.1162/evco_a_00229.

[95] Dua, Dheeru, and Casey Graff. *UCI Machine Learning Repository*. Available from `http://archive.ics.uci.edu/ml`.

[96] Tsanas, A. et al. Accurate telemonitoring of Parkinson's disease progression by noninvasive speech tests. *IEEE Transactions on Biomedical Engineering*. April, 2010, Vol. 57, No. 4, pp. 884–893.

[97] Le, Trang T et al. PMLB v1. 0: an open source dataset collection for benchmarking machine learning methods. *arXiv preprint arXiv:2012.00058*. 2020.

[98] Hamidieh, Kam. A data-driven statistical model for predicting the critical temperature of a superconductor. *Computational Materials Science*. 2018, Vol. 154, pp. 346–354. ISSN 0927-0256. Available from DOI 10.1016/j.commatsci.2018.07.052.

[99] Harris, Charles R. et al. Array programming with NumPy. *Nature*. Springer Science and Business Media LLC, Sep, 2020, Vol. 585, No. 7825, pp. 357–362. Available from DOI 10.1038/s41586-020-2649-2. Available from `https://doi.org/10.1038/s41586-020-2649-2`.

[100] Xianyi, Zhang, and Martin Kroeker. *OpenBLAS*. Available from `https://www.openblas.net/`. Accessed: 2021-04-06.

[101] Troise, Sarah Anne, and Thomas Helmuth. Lexicase Selection with Weighted Shuffle. In: Wolfgang Banzhaf et al., eds. *Genetic Programming Theory and Prac-*

tice XV. Cham: Springer International Publishing, 2018. pp. 89–104. ISBN 978-3-319-90512-9. Available from DOI 10.1007/978-3-319-90512-9_6.

[102] Helmuth, Thomas, Lee Spector, and James Matheson. Solving Uncompromising Problems With Lexicase Selection. *IEEE Transactions on Evolutionary Computation.* 2015, Vol. 19, No. 5, pp. 630–643. Available from DOI 10.1109/TEVC.2014.2362729.

[103] Buşoniu, Lucian, Damien Ernst, Bart De Schutter, and Robert Babuška. Approximate dynamic programming with a fuzzy parameterization. *Automatica.* 2010, Vol. 46, No. 5, pp. 804–814. ISSN 0005-1098. Available from DOI 10.1016/j.automatica.2010.02.006.

[104] Kubalík, Jiří, Eduard Alibekov, and Robert Babuška. Optimal Control via Reinforcement Learning with Symbolic Policy Approximation. *IFAC-PapersOnLine.* 2017, Vol. 50, No. 1, pp. 4162–4167. ISSN 2405-8963. Available from DOI 10.1016/j.ifacol.2017.08.805. 20th IFAC World Congress.

Appendix A
List of abbreviations

ANN ■ Artifical Neural Network
API ■ Application Programming Interface
CE ■ Coevolution
CGP ■ Cartesian Genetic Programming
DSR ■ Deep Symbolic Regression
EA ■ Evolutionary Algorithm
EDO ■ Evolutionary Dynamic Optimisation
EFS ■ Evolutionary Feature Synthesis
EoR-R^2 ■ End of Run R^2 score
ERC ■ Ephemeral Random Constant
FFNN ■ Feed-Forward Neural Network
FFX ■ Fast Feature eXtraction
GA ■ Genetic Algorithm
GLM ■ Generalized Linear Model
GP ■ Genetic Programming
GPTIPS ■ Genetic Programming Toolbox for the Identification of Physical Systems
GSGP ■ Geometric Semantic Genetic Programming
GSGP-Red ■ GSGP with Reduced trees
GUI ■ Graphical User Interface
KP ■ Kaizen Programming
LASSO ■ Least Absolute Shrinkage and Selection Operator
LCBF ■ Linear Combination of Basis Functions
LCF ■ Linear Combination of Features
LR ■ Linear Regression
LS ■ Local search
MAE ■ Mean Absolute Error
MGGP ■ Multi-Gene Genetic Programming
ML ■ Machine Learning
MRGP ■ Multiple Regression Genetic Programming
MSE ■ Mean Squared Error
MWUT ■ Mann-Whitney U-test
PGE ■ Prioritized Grammar Enumeration
RAUC ■ Relative Area Under Curve
RBF ■ Radial Basis Function
RF ■ Random Forest
RL ■ Reinforcement Learning
RMSE ■ Root Mean Squared Error
RNN ■ Recurrent Neural Network
SGD ■ Stochastic Gradient Descent
SNGP ■ Single-Node Genetic Programming

SR	■	Symbolic Regression
SVM	■	Support Vector Machine
SVR	■	Support Vector Regression
UPDRS	■	Unified Parkinson's Disease Rating Scale
VIA	■	Value Iteration Algorithm

Appendix B
RL benchmark data generation

The data for the VIA benchmarks (see Section 6.6.2) are generated by Fuzzy VIA [103–104] with triangular membership functions. The transition model f and the reward function ρ are given:

$$x_{k+1} = f(x_k, u_k) \tag{B.1}$$

$$r_{k+1} = \rho(x_k, u_k, x_{k+1}) \tag{B.2}$$

Next, a set $C = c_1, ..., c_N$ of points distributed over a regular grid in the state space is defined. Also a vector of triangular membership functions $\phi = [\phi_1(x), ..., \phi_N(x)]^T$ is defined such that each $\phi_i(x)$ is centred (its maximum is) at c_i and is zero at all other points, i.e. $\phi_j(c_i) = 0, \forall j \neq i$. Finally let there be a finite set of discrete control input values $U = u_1, ...u_M$. The V-function is then approximated as

$$V(x) = \theta^\top \phi(x) \tag{B.3}$$

where $\theta = [\theta_1, ..., \theta_N] \in \mathbf{R}^N$ is a vector of parameters found by iterating

$$\theta_i \leftarrow \max_{u \in U} \left[\rho(c_i, u) + \gamma\theta^\top \phi_i(f(c_i, u))\right] \tag{B.4}$$

Rewritten as an algorithm, this procedure can be seen in Algorithm B.1.

```
1  θ_0 ← 0_N
2  repeat in every iteration ℓ = 0, 1, 2, · · ·
3  |  for i = 1, · · · , N do
4  |  |  θ_{ℓ+1,i} ← max_{u∈U} [ρ(c_i, u) + γθ_ℓ^⊤ φ_i(f(c_i, u))]
5  |  end
6  until ‖θ_{l+1} − θ_l‖_∞ ≤ ε_{VI}
7  output θ̂* = θ_{ℓ+1}
```

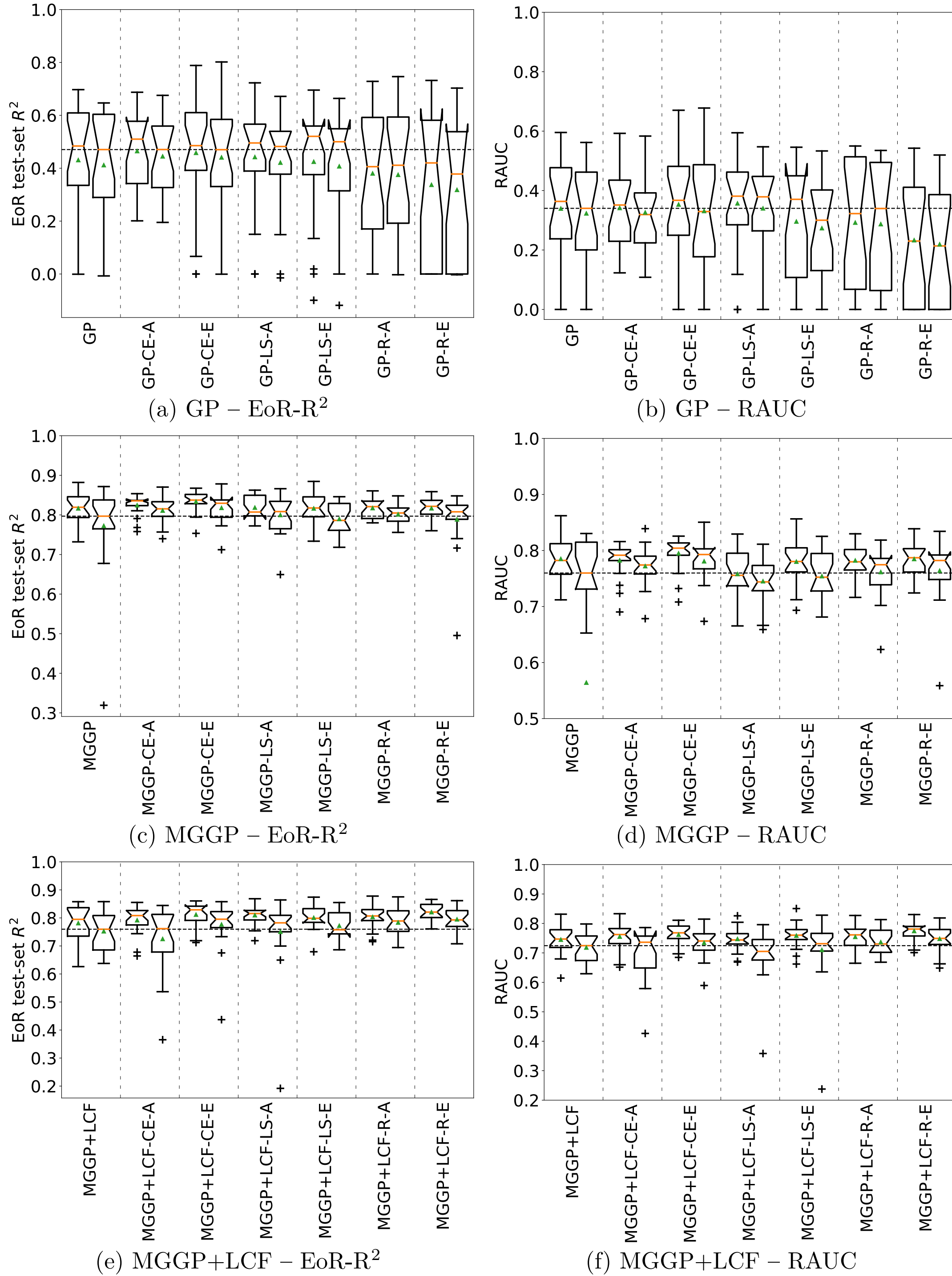
Algorithm B.1. Fuzzy V-iteration.

For the benchmarks used in Chapter 6, all the state points were identical to the centers of the membership functions so the V-function is determined by the values of θ_ℓ only. To generate the consecutive stages, not only the final $\hat{\theta}^*$ is used but also all the intermediate values of θ_ℓ that form these stages.

Appendix C
Detailed performance plots of fitness prediction approaches

C.1 Regular experiments

Boxplots of results from Section 7.4.7 (figures start on the next page).



(a) GP – EoR-R^2 (b) GP – RAUC

(c) MGGP – EoR-R^2 (d) MGGP – RAUC

(e) MGGP+LCF – EoR-R^2 (f) MGGP+LCF – RAUC

Figure C.1. Boxplots of results on the ASN dataset. Subfigures (a) and (b) show results for the GP base algorithm, subfigures (c) and (d) show results for the MGGP base algorithm, and subfigures (e) and (f) show results for the MGGP+LCF base algorithm. Subfigures (a), (c), and (e) show the EoR-R^2 score, while subfigures (b), (d), and (f) show the relative area under curve. Each configuration has a pair of boxplots, where the left one shows the training data performance and the right one shows the testing data performance. *Note*: for the sake of clarity, a few outliers are cropped from the plots – one in subfigure (d): for test-set MGGP at -3.25; two in subfigure (f): for test-set MGGP+LCF-CE-A at -247586.24 and test-set MGGP+LCF-LS-A at -60.37 and -5.53.

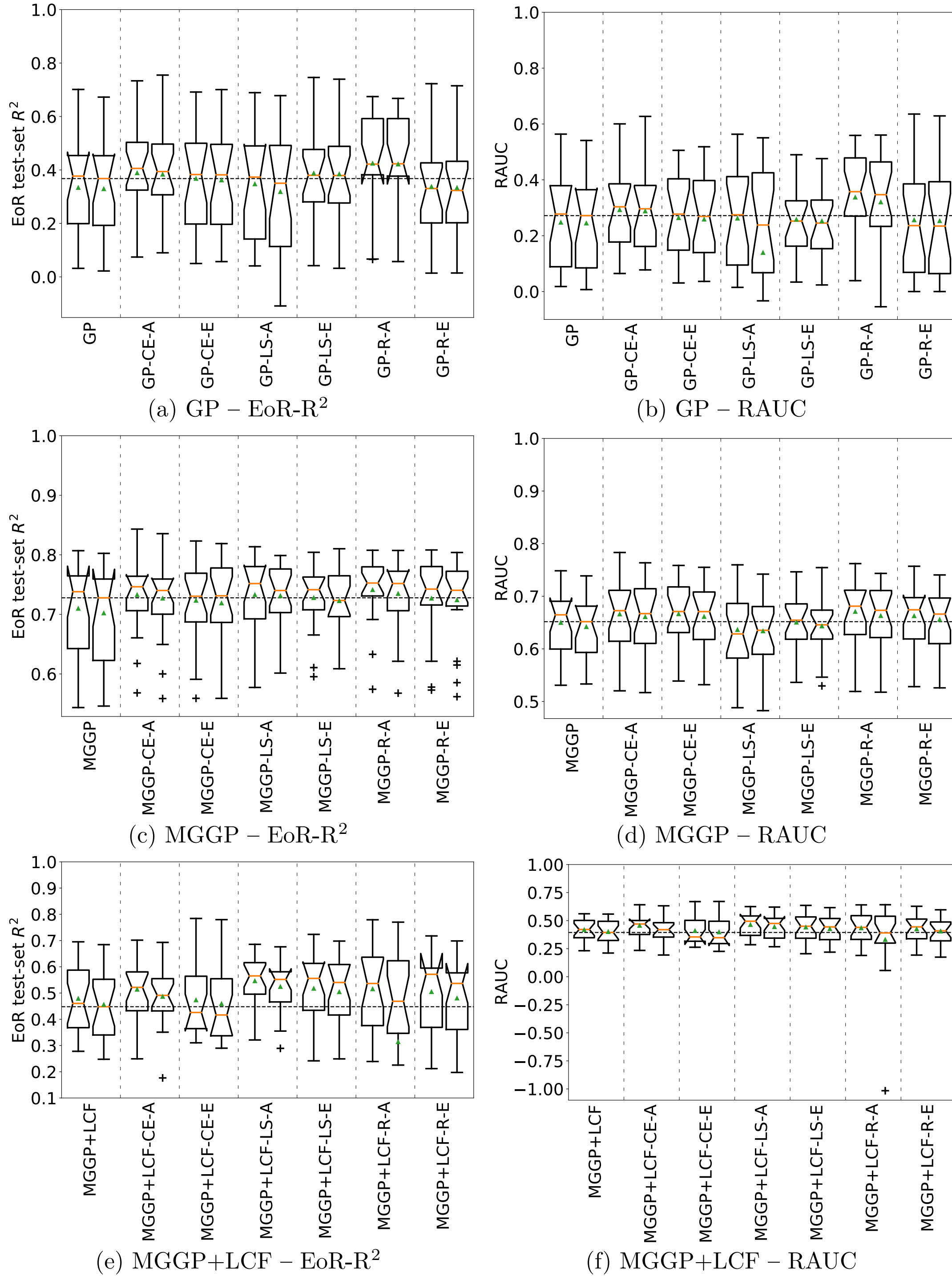
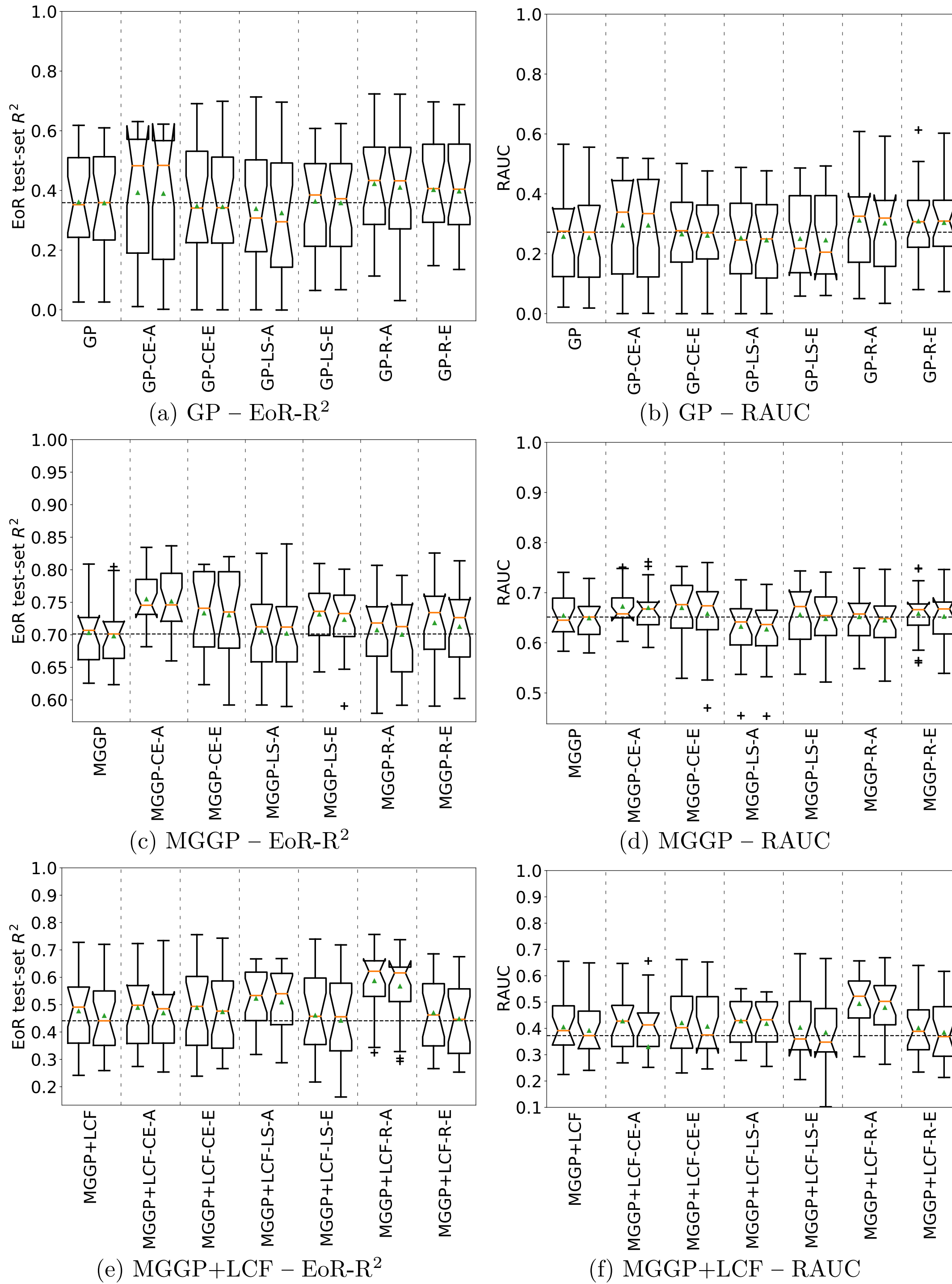


Figure C.2. Boxplots of results on the ParkinsonMotor dataset. Subfigures (a) and (b) show results for the GP base algorithm, subfigures (c) and (d) show results for the MGGP base algorithm, and subfigures (e) and (f) show results for the MGGP+LCF base algorithm. Subfigures (a), (c), and (e) show the EoR-R^2 score, while subfigures (b), (d), and (f) show the relative area under curve. Each configuration has a pair of boxplots, where the left one shows the training data performance and the right one shows the testing data performance. *Note*: for the sake of clarity, a few outliers are cropped from the plots – one in subfigure (b) for test-set GP-LS-A at -2.08, one in subfigure (e) for test-set MGGP+LCF-R-A at -3.10, and one in subfigure (f) for test-set MGGP+LCF-CE-A at -318544.15.

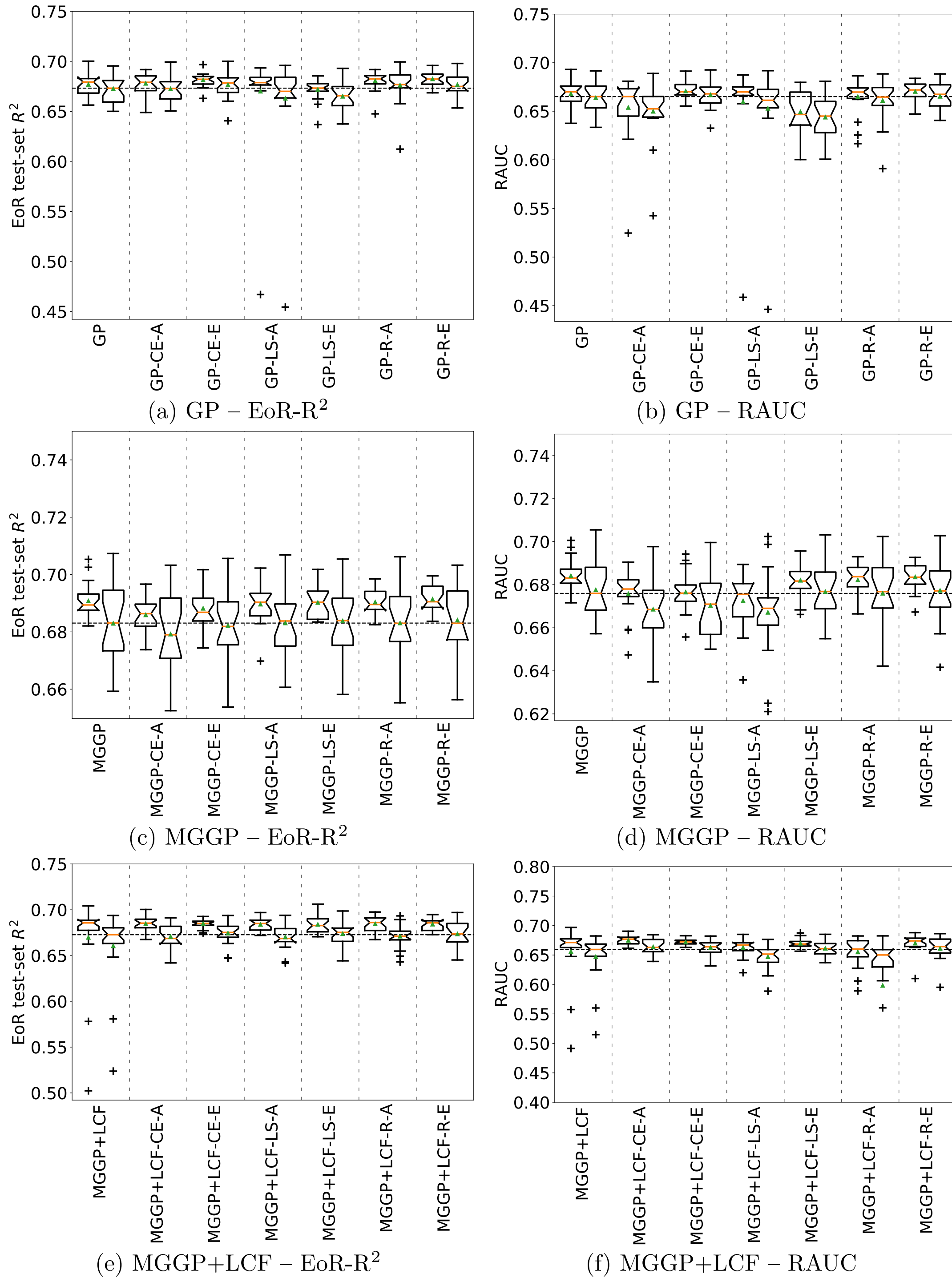


(a) GP – EoR-R^2 (b) GP – RAUC

(c) MGGP – EoR-R^2 (d) MGGP – RAUC

(e) MGGP+LCF – EoR-R^2 (f) MGGP+LCF – RAUC

Figure C.3. Boxplots of results on the ParkinsonTotal dataset. Subfigures (a) and (b) show results for the GP base algorithm, subfigures (c) and (d) show results for the MGGP base algorithm, and subfigures (e) and (f) show results for the MGGP+LCF base algorithm. Subfigures (a), (c), and (e) show the EoR-R^2 score, while subfigures (b), (d), and (f) show the relative area under curve. Each configuration has a pair of boxplots, where the left one shows the training data performance and the right one shows the testing data performance. *Note*: for the sake of clarity, a few outliers are cropped from the plots – one in subfigure (f) for test-set MGGP+LCF-CE-A at -1.43.



(a) GP – EoR-R^2

(b) GP – RAUC

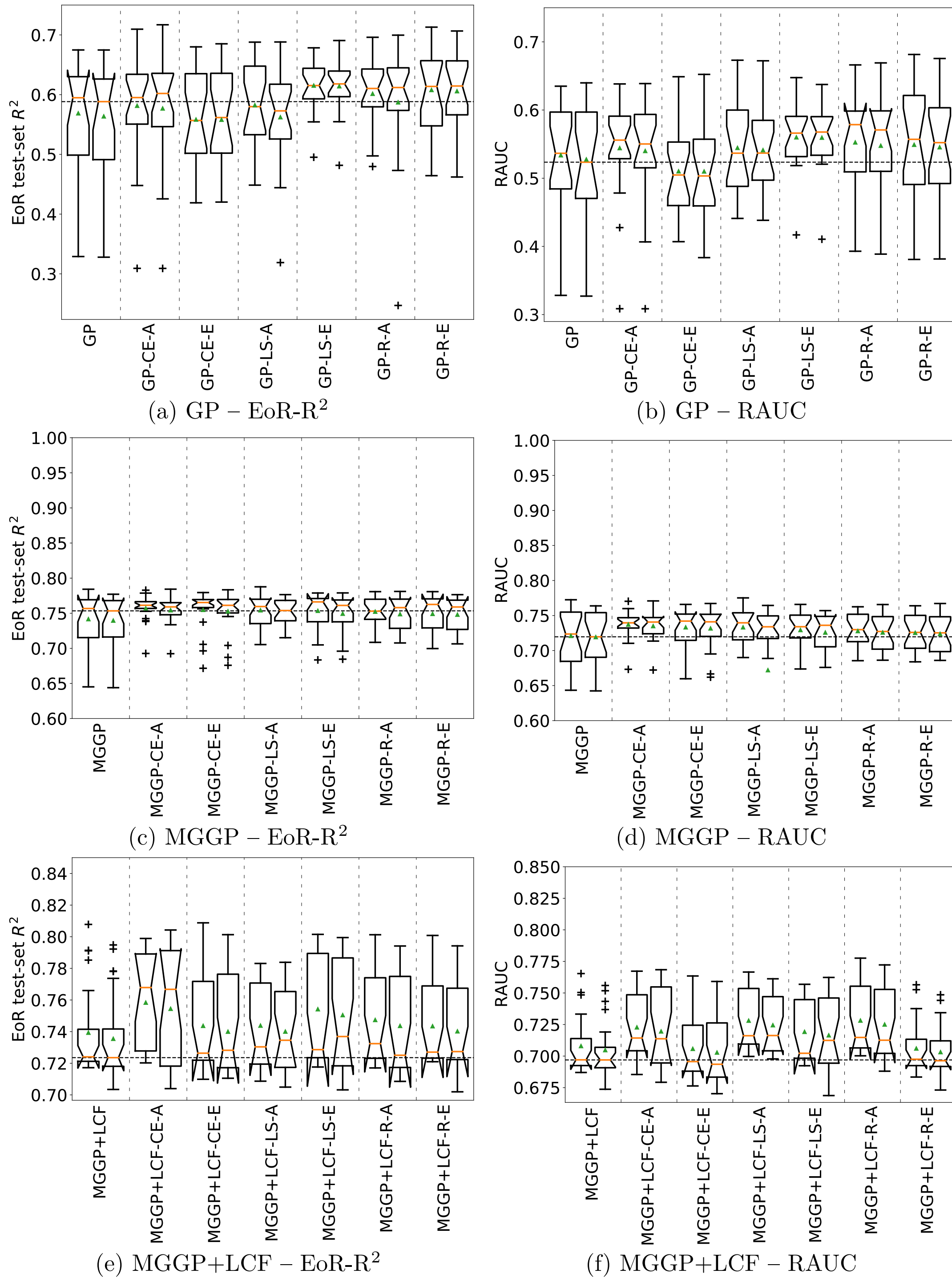
(c) MGGP – EoR-R^2

(d) MGGP – RAUC

(e) MGGP+LCF – EoR-R^2

(f) MGGP+LCF – RAUC

Figure C.4. Boxplots of results on the puma8NH dataset. Subfigures (a) and (b) show results for the GP base algorithm, subfigures (c) and (d) show results for the MGGP base algorithm, and subfigures (e) and (f) show results for the MGGP+LCF base algorithm. Subfigures (a), (c), and (e) show the EoR-R^2 score, while subfigures (b), (d), and (f) show the relative area under curve. Each configuration has a pair of boxplots, where the left one shows the training data performance and the right one shows the testing data performance. *Note*: for the sake of clarity, a few outliers are cropped from the plots – one in subfigure (f) for test-set MGGP+LCF-R-A at -0.26.



(a) GP – EoR-R^2

(b) GP – RAUC

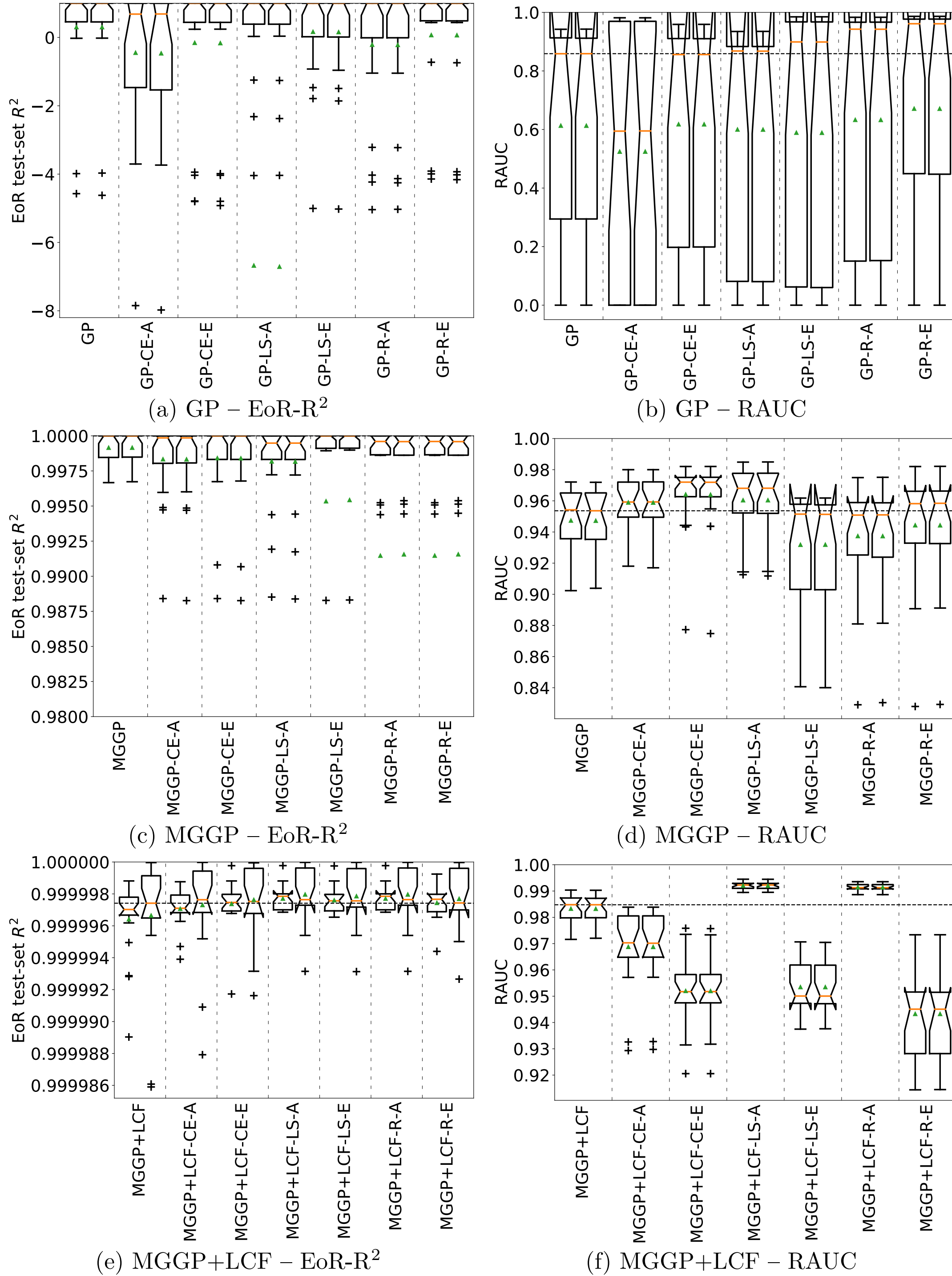
(c) MGGP – EoR-R^2

(d) MGGP – RAUC

(e) MGGP+LCF – EoR-R^2

(f) MGGP+LCF – RAUC

Figure C.5. Boxplots of results on the SupCon dataset. Subfigures (a) and (b) show results for the GP base algorithm, subfigures (c) and (d) show results for the MGGP base algorithm, and subfigures (e) and (f) show results for the MGGP+LCF base algorithm. Subfigures (a), (c), and (e) show the EoR-R^2 score, while subfigures (b), (d), and (f) show the relative area under curve. Each configuration has a pair of boxplots, where the left one shows the training data performance and the right one shows the testing data performance. *Note*: for the sake of clarity, a few outliers are cropped from the plots – two in subfigure (c) for test-set MGGP-LS-A at -4.44, two in subfigure (d) for test-set MGGP-LS-A at -0.48, and one in subfigure (f) for test-set MGGP+LCF-LS-A at -2.40.



(a) GP – EoR-R^2

(b) GP – RAUC

(c) MGGP – EoR-R^2

(d) MGGP – RAUC

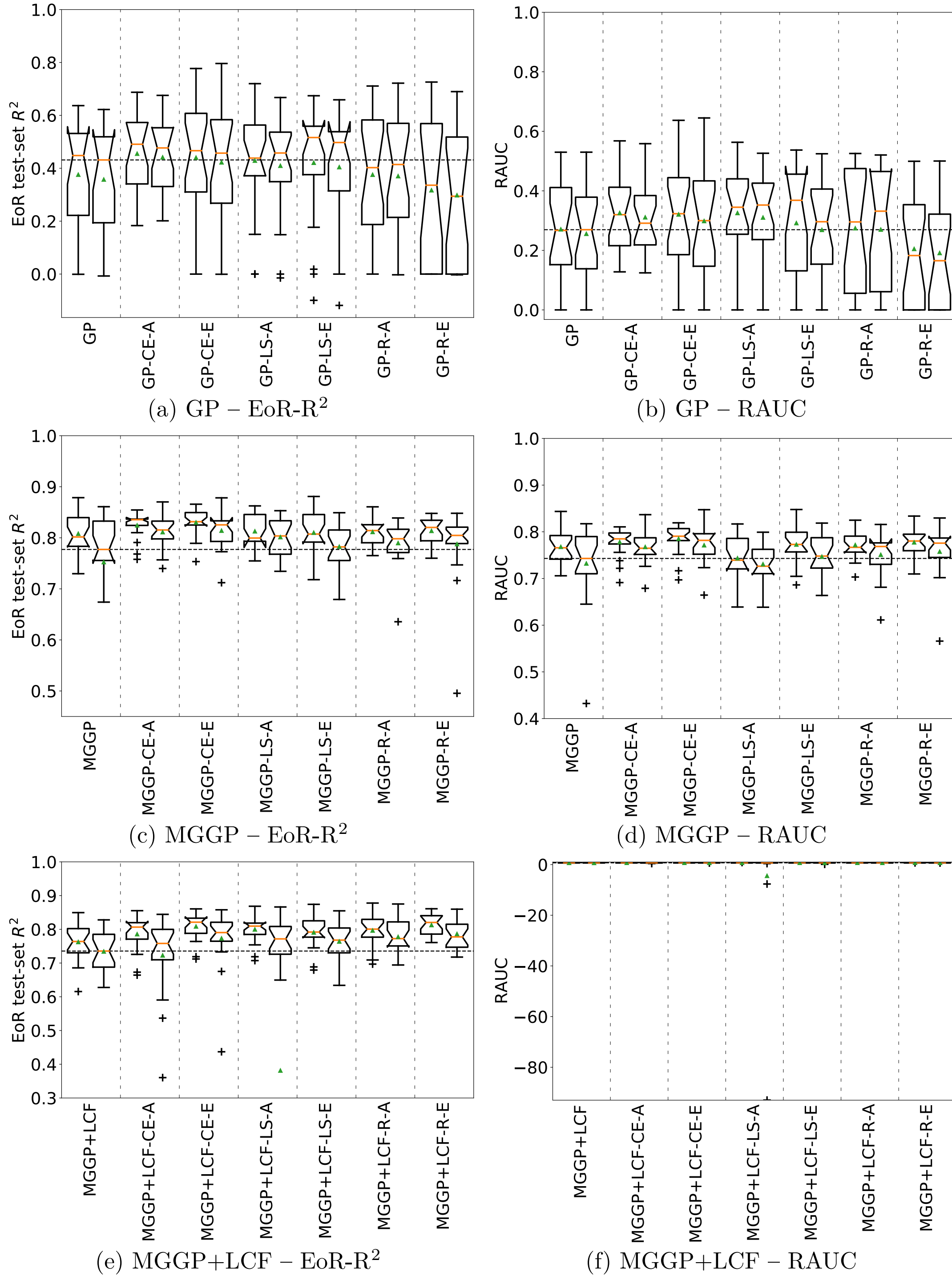
(e) MGGP+LCF – EoR-R^2

(f) MGGP+LCF – RAUC

Figure C.6. Boxplots of results on the WEC-A dataset. Subfigures (a) and (b) show results for the GP base algorithm, subfigures (c) and (d) show results for the MGGP base algorithm, and subfigures (e) and (f) show results for the MGGP+LCF base algorithm. Subfigures (a), (c), and (e) show the EoR-R^2 score, while subfigures (b), (d), and (f) show the relative area under curve. Each configuration has a pair of boxplots, where the left one shows the training data performance and the right one shows the testing data performance. *Note*: for the sake of clarity, a few outliers are cropped from the plots – two in subfigure (a): for train-set and test-set GP-LS-A at -140.74 and -140.12 respectively; six in subfigure (c): for train-set and test-set MGGP-R-E both at 0.85, for train-set and test-set MGGP-R-A both at 0.85, and for train-set and test-set MGGP-LS-E at 0.92 and 0.93 respectively.

C.2 Experiments with simulated time-demanding evaluation

Boxplots of results from Section 7.5.1 (figures start on the next page).

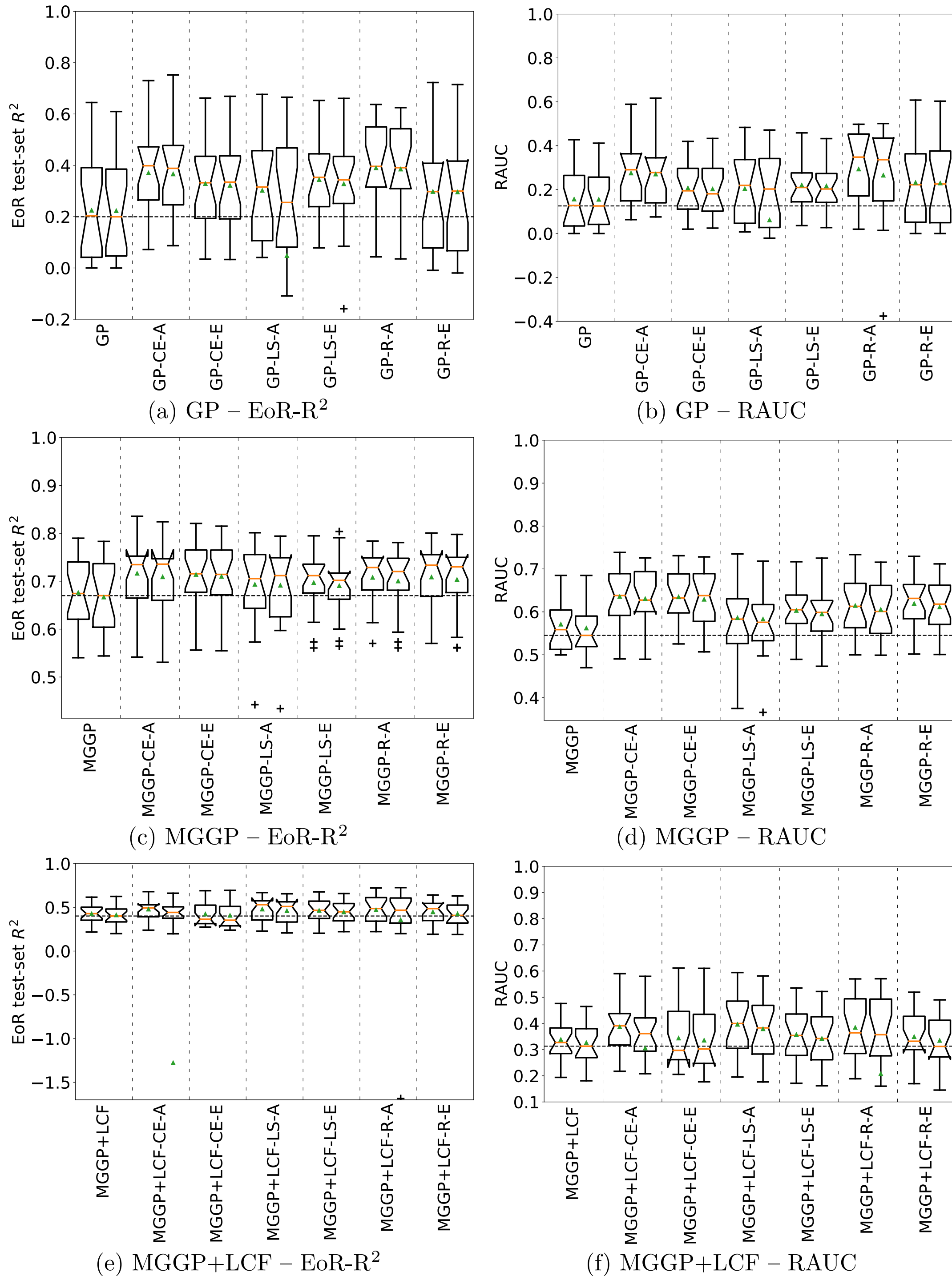


(a) GP – EoR-R^2 (b) GP – RAUC

(c) MGGP – EoR-R^2 (d) MGGP – RAUC

(e) MGGP+LCF – EoR-R^2 (f) MGGP+LCF – RAUC

Figure C.7. Boxplots of results on the ASN dataset for experiments with simulated time-demanding evaluation. Subfigures (a) and (b) show results for the GP base algorithm, subfigures (c) and (d) show results for the MGGP base algorithm, and subfigures (e) and (f) show results for the MGGP+LCF base algorithm. Subfigures (a), (c), and (e) show the EoR-R^2 score, while subfigures (b), (d), and (f) show the relative area under curve. Each configuration has a pair of boxplots, where the left one shows the training data performance and the right one shows the testing data performance. *Note*: for the sake of clarity, a few outliers are cropped from the plots – one in subfigure (c): for test-set MGGP at 0.09; one in subfigure (e): for test-set MGGP+LCF-LS-A at -7.02; one in subfigure (f): for test-set MGGP+LCF-CE-A at -283340.17.

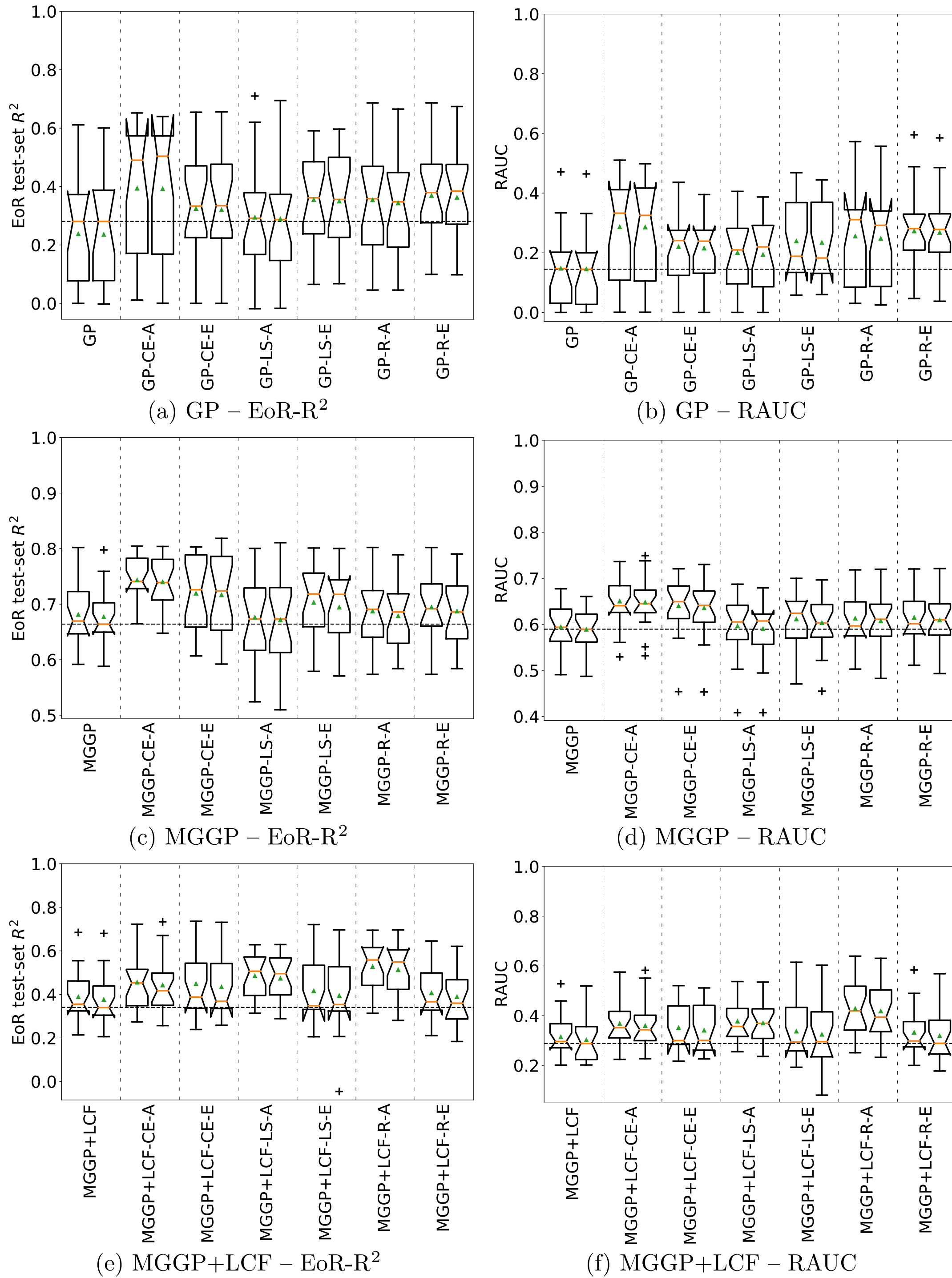


(a) GP – EoR-R^2 (b) GP – RAUC

(c) MGGP – EoR-R^2 (d) MGGP – RAUC

(e) MGGP+LCF – EoR-R^2 (f) MGGP+LCF – RAUC

Figure C.8. Boxplots of results on the ParkinsonMotor dataset for experiments with simulated time-demanding evaluation. Subfigures (a) and (b) show results for the GP base algorithm, subfigures (c) and (d) show results for the MGGP base algorithm, and subfigures (e) and (f) show results for the MGGP+LCF base algorithm. Subfigures (a), (c), and (e) show the EoR-R^2 score, while subfigures (b), (d), and (f) show the relative area under curve. Each configuration has a pair of boxplots, where the left one shows the training data performance and the right one shows the testing data performance. *Note*: for the sake of clarity, a few outliers are cropped from the plots – one in subfigure (a): for test-set GP-LS-A at -4.53; one in subfigure (b): for test-set GP-LS-A at -2.58; one in subfigure (e): for test-set MGGP+LCF-CE-A at -34.13; two in subfigure (f): for test-set MGGP+LCF-R-A at -3.09 and for test-set MGGP+LCF-CE-A at -0.90.

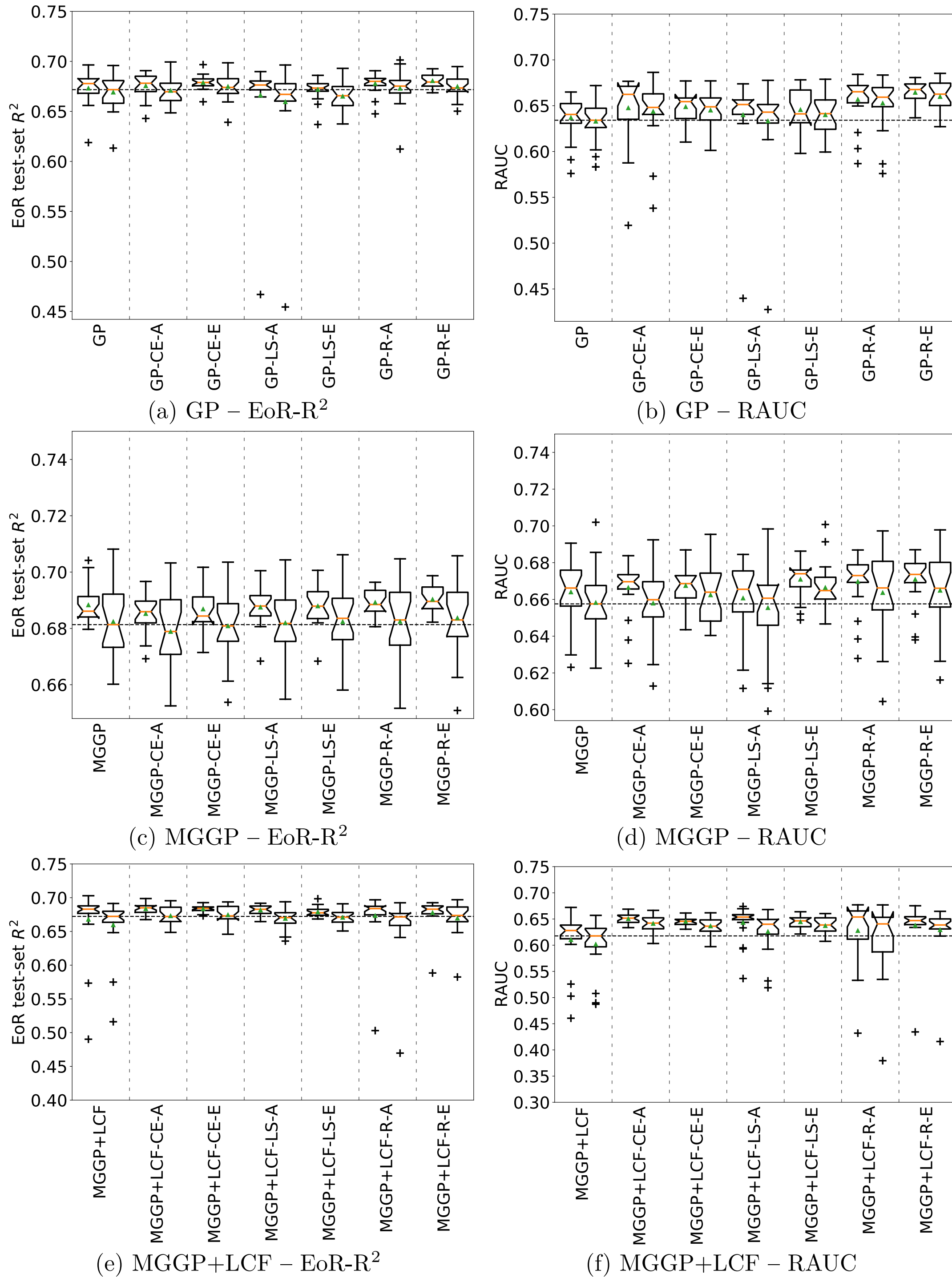


(a) GP – EoR-R^2 (b) GP – RAUC

(c) MGGP – EoR-R^2 (d) MGGP – RAUC

(e) MGGP+LCF – EoR-R^2 (f) MGGP+LCF – RAUC

Figure C.9. Boxplots of results on the ParkinsonTotal dataset for experiments with simulated time-demanding evaluation. Subfigures (a) and (b) show results for the GP base algorithm, subfigures (c) and (d) show results for the MGGP base algorithm, and subfigures (e) and (f) show results for the MGGP+LCF base algorithm. Subfigures (a), (c), and (e) show the EoR-R^2 score, while subfigures (b), (d), and (f) show the relative area under curve. Each configuration has a pair of boxplots, where the left one shows the training data performance and the right one shows the testing data performance.



(a) GP – EoR-R^2 (b) GP – RAUC

(c) MGGP – EoR-R^2 (d) MGGP – RAUC

(e) MGGP+LCF – EoR-R^2 (f) MGGP+LCF – RAUC

Figure C.10. Boxplots of results on the puma8NH dataset for experiments with simulated time-demanding evaluation. Subfigures (a) and (b) show results for the GP base algorithm, subfigures (c) and (d) show results for the MGGP base algorithm, and subfigures (e) and (f) show results for the MGGP+LCF base algorithm. Subfigures (a), (c), and (e) show the EoR-R^2 score, while subfigures (b), (d), and (f) show the relative area under curve. Each configuration has a pair of boxplots, where the left one shows the training data performance and the right one shows the testing data performance. *Note*: for the sake of clarity, a few outliers are cropped from the plots – one in subfigure (e): for test-set MGGP+LCF-R-A at -198.22; one in subfigure (f): for test-set MGGP+LCF-R-A at -6.73.

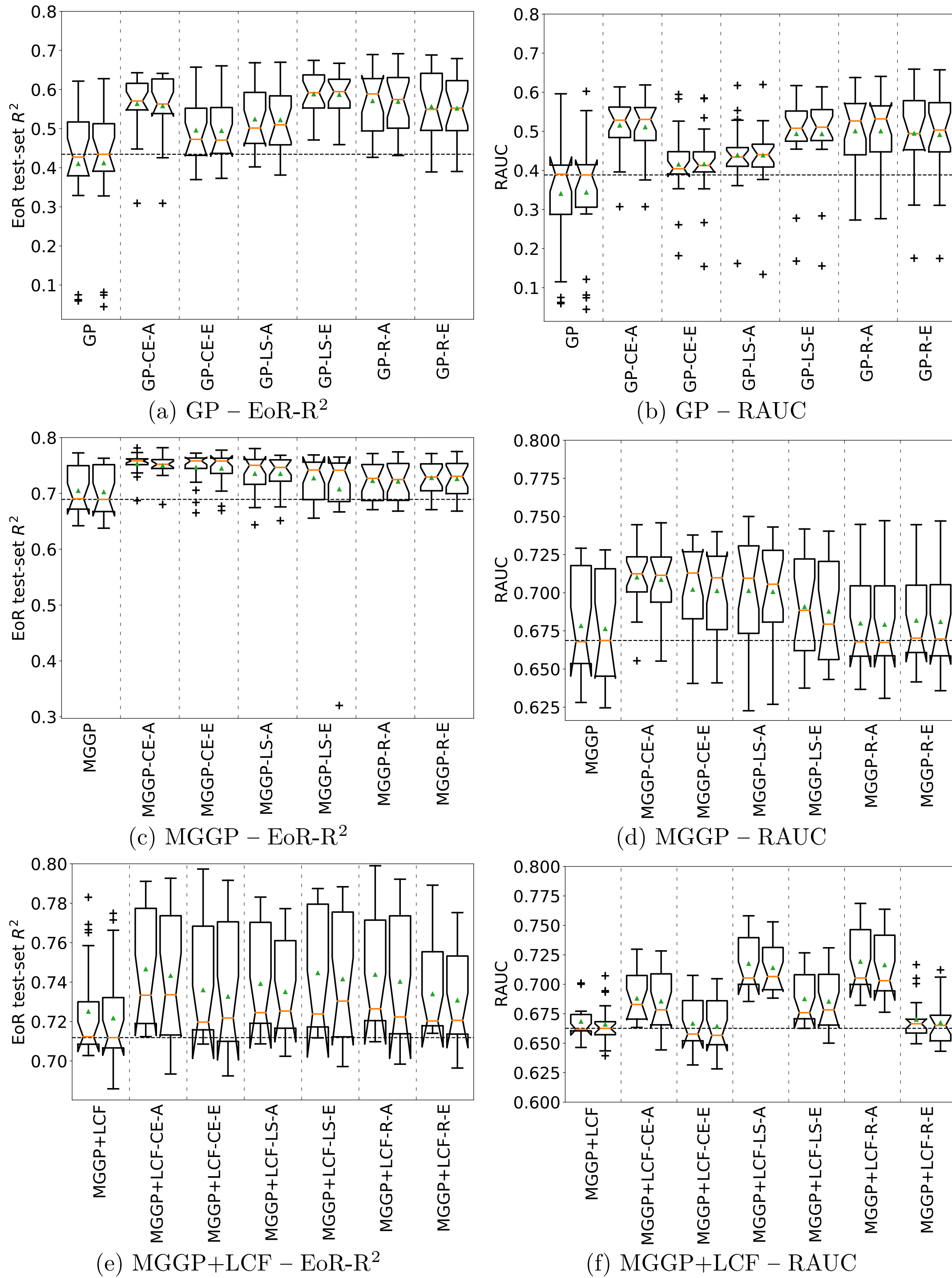
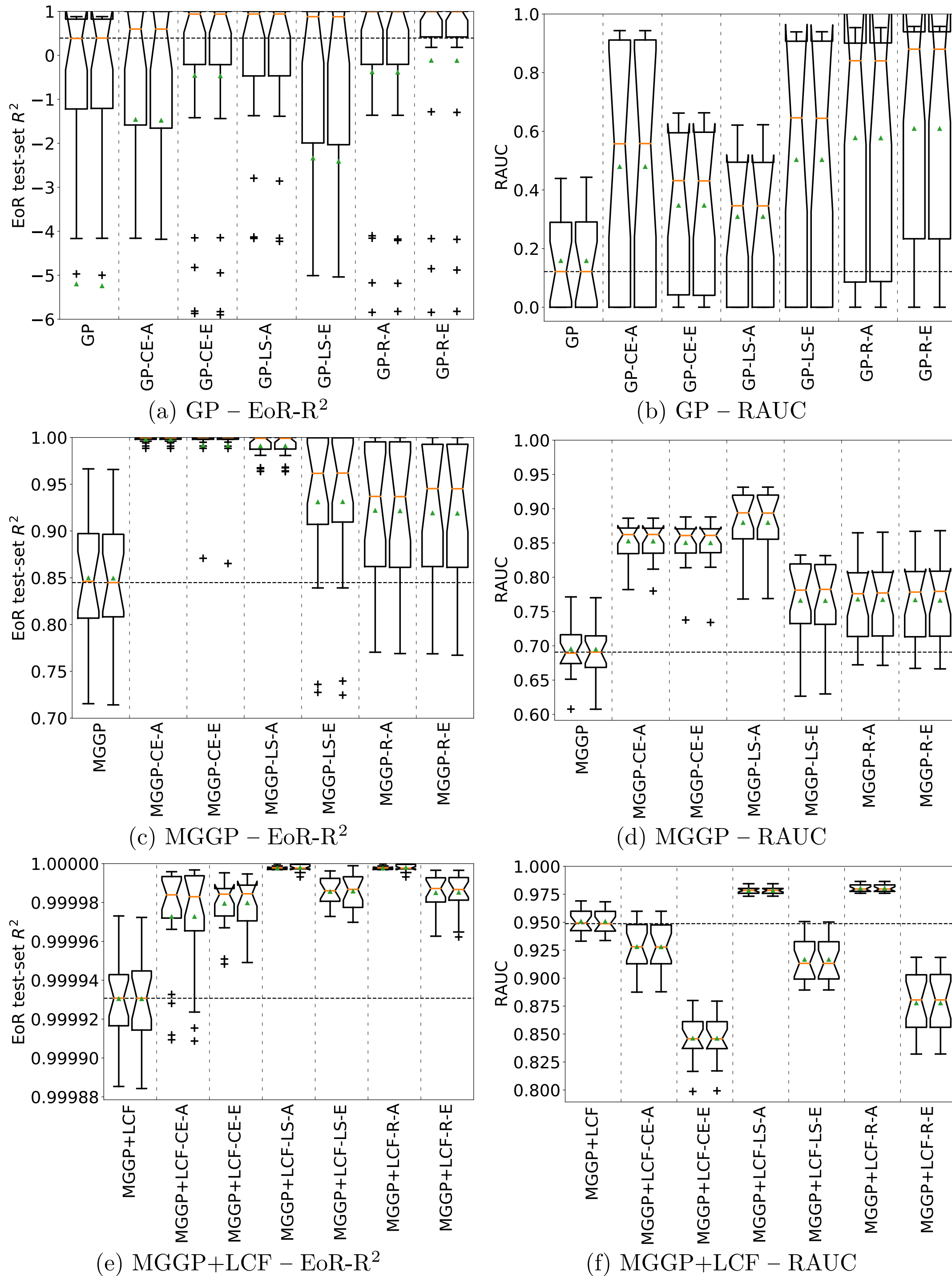


Figure C.11. Boxplots of results on the SupCon dataset for experiments with simulated time-demanding evaluation. Subfigures (a) and (b) show results for the GP base algorithm, subfigures (c) and (d) show results for the MGGP base algorithm, and subfigures (e) and (f) show results for the MGGP+LCF base algorithm. Subfigures (a), (c), and (e) show the EoR-R^2 score, while subfigures (b), (d), and (f) show the relative area under curve. Each configuration has a pair of boxplots, where the left one shows the training data performance and the right one shows the testing data performance. *Note*: for the sake of clarity, a few outliers are cropped from the plots – two in subfigure (c) for test-set MGGP-LS-E at 0.32.



(a) GP – EoR-R^2 (b) GP – RAUC

(c) MGGP – EoR-R^2 (d) MGGP – RAUC

(e) MGGP+LCF – EoR-R^2 (f) MGGP+LCF – RAUC

Figure C.12. Boxplots of results on the WEC-A dataset for experiments with simulated time-demanding evaluation. Subfigures (a) and (b) show results for the GP base algorithm, subfigures (c) and (d) show results for the MGGP base algorithm, and subfigures (e) and (f) show results for the MGGP+LCF base algorithm. Subfigures (a), (c), and (e) show the EoR-R^2 score, while subfigures (b), (d), and (f) show the relative area under curve. Each configuration has a pair of boxplots, where the left one shows the training data performance and the right one shows the testing data performance. *Note*: for the sake of clarity, a few outliers are cropped from the plots – eight in subfigure (a): for train-set and test-set GP-LS-A at -150.35 and -151.05 respectively, for train-set and test-set GP at -96.87 and -97.65 respectively, for train-set and test-set GP-LS-E at -41.85 and -43.10 respectively, and for train-set and test-set GP-CE-A at -25.84 and -26.01 respectively.

Appendix D
Author's publications

The author of this thesis is highlighted in bold. The "coauthorship" field describes the thesis author's proportion of authorship of the given publication. Number of citations[1] from Web of Science, Scopus, and Google Scholar are listed as of 20th July 2021.

D.1 Publications related to the thesis topic

D.1.1 Journal publications

[J1] **Žegklitz, J.** and P. Pošík. Benchmarking state-of-the-art symbolic regression algorithms. *Genetic Programming and Evolvable Machines*. 2021, Vol. 22, No. 1, pp. 5–33. Available from DOI 10.1007/s10710-020-09387-0.
Coauthorship: 70 %; WoS: 2, Scopus: 1, Google Scholar: 7

[J2] **Žegklitz, J.** and P. Pošík, Symbolic regression in dynamic scenarios with gradually changing targets. *Applied Soft Computing*. 2019, Vol. 83. ISSN 1568-4946. Available from DOI 10.1016/j.asoc.2019.105621.
Coauthorship: 70 %; WoS: ∅, Scopus: 0, Google Scholar: 0

D.1.2 Conference and workshop publications

[C1] **Žegklitz, J.** and Petr Pošík. Combining Subtree and Ripple Crossover in Grammatical Evolution. In: *Proceedings of the 14th conference ITAT 2014 – Workshops and Posters*. Demänovská Dolina, Slovakia: Institute of Computer Science AS CR, 2014. pp. 106–111. ISBN 978-80-87136-19-5. Available from `http://artax.karlin.mff.cuni.cz/ bajel3am/itat2014/local/106_Zegklitz.pdf`.
Coauthorship: 73 %; WoS: ∅, Scopus: ∅, Google Scholar: 0

[C2] **Žegklitz, J.** and Petr Pošík. Symbolic Regression by Grammar-Based Multi-Gene Genetic Programming. In: *Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation*. New York, NY, USA: ACM, 2015. pp. 1217–1220. GECCO Companion '15. ISBN 978-1-4503-3488-4. Available from DOI 10.1145/2739482.2768484.
Coauthorship: 90 %; WoS: 2, Scopus: 3, Google Scholar: 3

[C3] **Žegklitz, J.** and Petr Pošík. Model Selection and Overfitting in Genetic Programming: Empirical Study. In: *Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation*. New York, NY, USA: ACM, 2015. pp. 1527–1528. GECCO Companion '15. ISBN 978-1-4503-3488-4. Available from DOI 10.1145/2739482.2764678.
Coauthorship: 75 %; WoS: 5, Scopus: 6, Google Scholar: 9

[1] Number "0" means that the publication is indexed by the respective database but records no citations. The symbol "∅" means that the publication is not indexed by the database.

[C4] Kubalík, J., E. Alibekov, **J. Žegklitz**, and R. Babuška, Hybrid Single Node Genetic Programming for Symbolic Regression. In: Ngoc Thanh Nguyen, Ryszard Kowalczyk, and Joaquim Filipe, eds. *Transactions on Computational Collective Intelligence XXIV*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016. pp. 61–82. ISBN 978-3-662-53525-7. Available from DOI 10.1007/978-3-662-53525-7_4.
Coauthorship: 25 %; WoS: 6, Scopus: 5, Google Scholar: 6

[C5] **Žegklitz, J.** and Petr Pošík. Linear Combinations of Features as Leaf Nodes in Symbolic Regression. In: *Proceedings of the Genetic and Evolutionary Computation Conference Companion*. New York, NY, USA: ACM, 2017. pp. 145–146. GECCO '17. ISBN 978-1-4503-4939-0. Available from DOI 10.1145/3067695.3076009.
Coauthorship: 90 %; WoS: 3, Scopus: ∅, Google Scholar: 5

[C6] **Žegklitz, J.** and Petr Pošík. Sequential Model Building in Symbolic Regression. In: *ITAT 2019: Information Technologies – Applications and Theory*. 2019.
Coauthorship: 85 %; WoS: ∅, Scopus: 0, Google Scholar: 0

D.1.3 Preprints and extended versions

[P1] **Žegklitz, J.** and Petr Pošík. Model Selection and Overfitting in Genetic Programming: Empirical Study [Extended Version]. (Extended version of [C3].) In: *arXiv preprint arXiv:1504.08168*. 2015. Available from `https://arxiv.org/abs/1504.08168`.
Coauthorship: 75 %; WoS: ∅, Scopus: ∅, Google Scholar: ∅

[P2] **Žegklitz, J.** and Petr Pošík. Symbolic Regression Algorithms with Built-in Linear Regression. (Preprint of [J1].) In: *arXiv preprint arXiv:1701.03641*. 2017. Available from `https://arxiv.org/abs/1701.03641`.
Coauthorship: 70 %; WoS: 4, Scopus: ∅, Google Scholar: 10

[P3] **Žegklitz, J.** and Petr Pošík. Learning Linear Feature Space Transformations in Symbolic Regression. (Extended version of [C5].) In: *arXiv preprint arXiv:1704.05134*. 2017. Available from `https://arxiv.org/abs/1704.05134`.
Coauthorship: 70 %; WoS: 2, Scopus: ∅, Google Scholar: 0

D.2 Publications not related to the thesis topic

D.2.1 Journal publications

[J4] Bemš, J., O. Starý, M. Macaš, **J. Žegklitz**, and P. Pošík, Innovative default prediction approach. *Expert Systems with Applications*. 2015, Vol. 42, No. 17, pp. 6277–6285. ISSN 0957-4174. Available from DOI 10.1016/j.eswa.2015.04.053.
Coauthorship: 10 %; WoS: 8, Scopus: 10, Google Scholar: 15

Appendix E
Curriculum Vitae

Jan Žegklitz was born in Prague, Czech Republic, in 1988. He started his bachelor studies in 2008 in the Software Technologies and Management study programme at Czech Technical University in Prague, and graduated in 2011 with praise. He continued his master's studies at the same faculty in the Open Informatics study programme, and graduated in 2013 with honour. After a half-year pause, he started pursuing his Ph.D. at the Department of Cybernetics in the area of genetic programming and symbolic regression.

During his research years, Jan has published two articles in impacted journals as the first author, and collaborated on one other impacted journal article.

Jan has participated in two research projects: (1) Symbolic Regression for Reinforcement Learning in Continuous Spaces, and (2) Metalearning for Extraction of Rules with Numerical Consequents, both referenced in the Acknowledgements. He also participated in the 2019 SIGEVO Summer School.

Jan led labs for several subjects at the university: Programming 2, Algorithms and programming, and Artificial Intelligence. He was also an opponent of three master's theses by Vladimír Klouček, Vojtěch Hejl, and Andrej Kudinov respectively.

Besides his academic work, he also worked in a software company that develops an advanced planning and scheduling system where he utilized his programming skills acquired during the studies.

....
Jan Žegklitz
Prague, July 2021