JOHANNES KEPLER
UNIVERSITÄT LINZ

Eingereicht von
Michael Kommenda

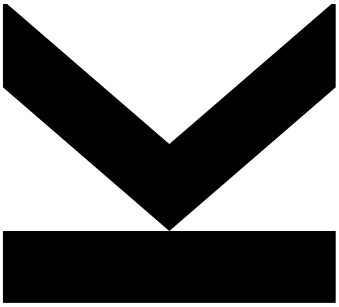
Angefertigt am
Institute for Formal Models and Verification

Betreuer und
Erstbeurteiler
FH-Prof. PD DI Dr. Michael Affenzeller

Zweitbeurteiler
a.Univ.-Prof. Dr. Josef Küng

April 2018

Local Optimization and Complexity Control for Symbolic Regression



Dissertation

zur Erlangung des akademischen Grades

Doktor der technischen Wissenschaften

im Doktoratsstudium

Technische Wissenschaften

JOHANNES KEPLER
UNIVERSITÄT LINZ
Altenbergerstraße 69
4040 Linz, Österreich
www.jku.at
DVR 0093696

Acknowledgments

Several people have been essential for this thesis to become a reality. First and foremost and I want to thank Michael Affenzeller for encouraging and giving me the possibility to pursue a doctorate within the research group Heuristic and Evolutionary Algorithms Laboratory (HEAL). Furthermore, I am sincerely grateful to Josef Küng for acting as an examiner without whom this endeavor would not be possible.

Research in general is rarely an effort by a single person and numerous people shaped the way I think about computer science, data-based modeling, algorithm design, experimentation, and optimization in general. In addition to my advisors, I want to especially thank Gabriel Kronberger, Stefan Wagner, Andreas Beham, and Stephan Winkler who challenged me constantly to improve and whose thoughts and discussions influenced the scientific contributions of this thesis.

Working within the research group HEAL has always been a great privilege and pleasure for me. Therefore, I want to thank all of my colleagues for making my work life so enjoyable. Finally and most importantly, I would like to thank my soon-to-be wife Romy for always supporting me.

Abstract

Symbolic regression is a data-based machine learning approach that creates interpretable prediction models in the form of mathematical expressions without the necessity to specify the model structure in advance. Due to numerous possible models, symbolic regression problems are commonly solved by metaheuristics such as genetic programming. A drawback of this method is that because of the simultaneous optimization of the model structure and model parameters, the effort for learning from the presented data is increased and the obtained prediction accuracy could suffer. Furthermore, genetic programming in general has to deal with bloat, an increase in model length and complexity without an accompanying increase in prediction accuracy, which hampers the interpretability of the models. The goal of this thesis is to develop and present new methods for symbolic regression, which improve prediction accuracy, interpretability, and simplicity of the models.

The prediction accuracy is improved by integrating local optimization techniques that adapt the numerical model parameters in the algorithm. Thus, the symbolic regression problem is divided into two separate subproblems: finding the most appropriate structure describing the data and finding optimal parameters for the specified model structure. Genetic programming excels at finding appropriate model structures, whereas the Levenberq-Marquardt algorithm performs least-squares curve fitting and model parameter tuning. The combination of these two methods significantly improves the prediction accuracy of generated models.

Another improvement is to turn the standard single-objective formulation of symbolic regression into a multi-objective one, where the prediction accuracy is maximized while the model complexity is simultaneously minimized. As a result the algorithm does not produce a single solution, but a Pareto front of models with varying accuracy and complexity. In addition, a novel complexity measure for multi-objective symbolic regression is developed that includes syntactic and semantic information about the models while still being efficiently computed. By using this new complexity measure the generated models get simpler and the occurrence of bloat is reduced.

Kurzfassung

Symbolische Regression ist ein datenbasiertes, maschinelles Lernverfahren bei dem Vorhersagemodelle in Form mathematischer Ausdrücke ohne vorgegebener Modellstruktur erstellt werden. Wegen der Vielzahl möglicher Modelle, welche die Daten beschreiben, werden symbolische Regressionsprobleme meist mittels genetischer Programmierung gelöst. Ein Nachteil dabei ist, dass wegen der gleichzeitigen Optimierung der Modellstruktur und deren Parameter, der Aufwand zum Lernen der Modelle erhöht ist und deren Genauigkeit verringert sein kann. Zusätzlich wird die Interpretierbarkeit der Modelle durch das Auftreten überflüssiger Ausdrücke (engl. *bloat*), welche die Modelle verkomplizieren ohne deren Genauigkeit zu erhöhen, erschwert. Das Ziel dieser Dissertation ist es neue Methoden zur Verbesserung der Genauigkeit und Interpretierbarkeit symbolischer Regressionsmodelle zu entwickeln.

Die Genauigkeit der Modelle wird durch die Integration lokaler Optimierung, welche die numerischen Parameter der Modelle anpasst, erhöht. Dadurch wird das Regressionsproblem in zwei Aufgaben unterteilt. Zuerst wird eine passende Modellstruktur identifiziert und anschließend deren numerischen Parameter adaptiert. Genetische Programmierung wird zur Identifikation der Modellstruktur verwendet, während der Levenberg-Marquardt Algorithmus eine nichtlineare Anpassung der numerischen Parameter vornimmt. Durchgeführte Experimente zeigen, dass die Kombination dieser Methoden in einer deutlichen Verbesserung der Modellgenauigkeit resultiert.

Die Interpretierbarkeit der Modelle wird durch eine Änderung der Problemformulierung von einzelkriterieller zu multikriterieller Optimierung verbessert, wodurch die Genauigkeit der Modelle maximiert während gleichzeitig deren Komplexität minimiert wird. Das Ergebnis ist somit nicht mehr ein einzelnes Modell, sondern eine Pareto-Front, welche den Kompromiss zwischen Genauigkeit und Komplexität widerspiegelt. Zusätzlich wird ein neues Komplexitätsmaß für symbolische Regression vorgestellt, welches syntaktische und semantische Informationen berücksichtigt. Durch den Einsatz dieses neuen Komplexitätsmaßes werden die erzeugten Modelle besser interpretierbar und überflüssige Ausdrücke vermieden.

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Dissertation selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.

Die vorliegende Dissertation ist mit dem elektronisch übermittelten Textdokument identisch.

Michael Kommenda, Linz, 2018

Contents

Chapter 1

Introduction

1.1 Motivation

The importance of data-based modeling techniques has risen with the vast increase of available data and computing power during the last years. Several methods such as linear regression [Draper et al., 1966], random forests [Breiman et al., 1984], support vector machines [Vapnik, 1999], artificial neural networks [Haykin, 1998], Gaussian processes [Rasmussen, 2004] have been developed to extract the most information from the data and build accurate regression models.

Symbolic regression is another data-based modeling technique that distinguishes itself from other methods due to the fact that the model structure need not be predefined, but is automatically adapted to the data. Another advantage is that the generated model is represented as mathematical expression that is open for inspection and interpretation [Affenzeller et al., 2014]. Symbolic regression problems are commonly solved by genetic programming, because the variable-length encoding of solutions in genetic programming is especially suited for representing mathematical expressions and due to the enormous search space heuristic methods are preferred.

Genetic programming is well suited to search for the model structure describing the data, but is not the very good in determining the appropriate numerical constants to fit the model structure to the data. As a result of inappropriate numerical values in the generated model, their prediction accuracy is reduced. This has also been noted by O'Neill et al. [2010], where a quote of John Koza is presented:

> ... the finding of numerical constants is a skeleton in the GP closet
> ... [and an] area of research that requires more investigation ...

Another issue regarding genetic programming for solving symbolic regression problems is that the algorithm is hard to configure and parameterize to achieve high quality solutions. Furthermore, not only the algorithm configuration affects the obtained quality, but also the restrictions imposed to the symbolic regression problem. These restriction include the mathematical functions occurring in the models; Is it allowed to use trigonometric functions during model creation or are only arithmetic functions usable?

In addition, it is common in most genetic programming systems to imposes a static size limit to the model length, because otherwise the models would grow infinitely and their interpretability would be severely hampered.

The major problem with these two restrictions that are enforced during modeling is that they are highly problem specific. They should be set as tight as possible without affecting the accuracy of the generated models. In general simpler models are preferred to complex ones as long as they generalize equally well. However, to achieve simple models by genetic programming several different size limits and function configurations have to be tested for the problem at hand, because appropriate values cannot be identified a-priori. These issues further complicate the use of symbolic regression and are primarily addressed in this thesis.

1.2 Main Research Contributions

The broad research topic of this thesis is to identify new ways for improving symbolic regression by addressing its weak points. Therefore, constants creation and the complexity of solutions are studied in detail and based on empirical observations improved methods are derived. This research has been performed by using tree-based genetic programming for symbolic regression and also the new methods are derived and tested with this optimization method. However, these improvements are not specific to tree-based genetic programming, but are applicable to any kind of algorithm solving symbolic regression problems.

All of the described experiments, methodological improvements, and analyses are implemented in and have been performed with the open-source framework for heuristic optimization HeuristicLab [Wagner et al., 2014].

Local Optimization

The main advancements for the creation and identification of constants for symbolic regression is a new local optimization technique termed constants optimization by nonlinear least squares (CO-NLS). CO-NLS is integrated in the genetic programming algorithm by adapting the numeric values in the models before a solution is evaluated. However, CO-NLS is more generally applicable and can be integrated in any algorithm and can be even utilized as a post-processing step to improve the accuracy of generated models. The detailed research contributions in the field of constants creation and local optimization are the following:

- Review of constants creation, constants adaptation, and local optimization methods in symbolic regression
- Comparison of different objective functions and linear scaling on benchmark problems
- Development and implementation of CO-NLS by combining automatic differentiation and gradient-based nonlinear least squares optimization
- Performance evaluation of CO-NLS in genetic programming and offspring selection genetic programming to highlight its advantages and disadvantages as well as the influence of its parameters

Complexity Control

The second main topic of this thesis is to investigate ways of creating simpler models. Therefore, the role of the tree length restriction in genetic programming is evaluated and its effect on the obtained solutions is presented. Afterwards more advanced complexity metrics for symbolic regression models are discussed and from these observations another complexity metric is derived. The newly defined *recursive complexity* is easily implemented and can be calculated with a single tree iterations without evaluating the models on the presented data. In contrast to other complexity metrics that share these benefits, it takes the semantics of the model into account. The recursive complexity does not provide the most accurate estimation of the model's complexity, but gives the algorithm a strong enough search direction so that simple models can be obtained. Furthermore, it becomes unnecessary to specify the mathematical functions that can occur in the models, because the appropriate ones are automatically detected.

The detailed research contributions to complexity control while generating symbolic regression solutions are the following:

- Review of complexity metrics and methods for complexity control in symbolic regression
- Definition and discussion of the recursive complexity and its parameterization for multi-objective symbolic regression
- Evaluation of the suitability of NSGA-II for performing multi-objective symbolic regression
- Adaptations of NSGA-II by changing the dominance criterion and discretizing objective functions
- Comparison of single and multi-objective symbolic regression on noise-free and noisy benchmark problems

1.3 Research Background

This methods and ideas contributing to this thesis have been mainly developed within the Josef Ressel-Center for heuristic optimization *Heureka!* and the K-Project *HOPL - Heuristic Optimization in Production and Logistics*, both supported by the Austrian Research Promotion Agency (FFG).

Parts of the work and ideas presented in this thesis have already appeared in previous publications by the author:

- M. Kommenda, G. K. Kronberger, M. Affenzeller, S. M. Winkler, C. Feilmayr, S. Wagner - Symbolic Regression with Sampling - 22nd European Modeling and Simulation Symposium EMSS 2010, Fes, Morocco, 2010, pp. 13-18
- M. Kommenda, G. K. Kronberger, S. Wagner, S. M. Winkler, M. Affenzeller - On the Architecture and Implementation of Tree-based Genetic Programming in HeuristicLab - Companion Publication of the 2012 Genetic and Evolutionary Computation Conference, GECCO'12 Companion, Philadelphia, United States of America, 2012, pp. 101-108
- M. Kommenda, M. Affenzeller, G. K. Kronberger, S. M. Winkler - Nonlinear Least Squares Optimization of Constants in Symbolic Regression - Computer Aided Systems Theory EUROCAST 2013, Las Palmas de Gran Canaria, Spain, 2013, pp. 420-427

- M. Kommenda, G. K. Kronberger, S. M. Winkler, M. Affenzeller, S. Wagner - Effects of Constant Optimization by Nonlinear Least Squares Minimization in Symbolic Regression - Companion Publication of the 2013 Genetic and Evolutionary Computation Conference, GECCO'13 Companion, Amsterdam, Netherlands, 2013, pp. 1121-1127
- M. Kommenda, A. Beham, M. Affenzeller, G. K. Kronberger - Complexity Measures for Multi-Objective Symbolic Regression - Proceedings of the International Conference on Computer Aided Systems Theory (EUROCAST 2015), Las Palmas, Gran Canaria, Spain, 2015
- M. Kommenda, G. Kronberger, M. Affenzeller, S.M. Winkler, B. Burlacu - Evolving Simple Symbolic Regression Models by Multi-objective Genetic Programming - Genetic Programming Theory and Practice XIII, (Editors: Rick Riolo, Jason H. Moore, Mark Kotanchek) - Springer, 2016

1.4 Chapter Overview

Chapter 2 introduces the problem of symbolic regression and discusses its differences to standard regression analysis. It is organized to introduce more generally applicable concepts first and focuses more and more on relevant topics for thesis. The most common approach to solve symbolic regression is genetic programming, which is discussed in Section 2.1. Starting with a general description of genetic programming and its historical development, the main concepts of the algorithm are detailed. Afterwards, tree-based genetic programming is presented, where solutions are represented as symbolic expression trees, and the connection between symbolic regression and genetic programming is established. Genetic programming and symbolic regression are already established techniques and in Section 2.2 popular frameworks and software to implement and perform symbolic regression are listed. Finally, this chapter is concluded by Section 2.3 that discusses two alternative approaches for deterministic symbolic regression that are not based on genetic programming.

Chapter 3 focuses on local optimization and starts by discussing the role of constants in symbolic regression. Section 3.1 introduces ephemeral random constants and how they are manipulated, constants handling in HeuristicLab, and the benefits of linear scaling. After this introduction to the topic, existing constants optimization approaches are discussed in Section 3.2. In Section 3.3 constants optimization by nonlinear least squares (CO-NLS) is presented, the implementation details are described, and first effects of including CO-NLS in genetic programming are visualized. An extensive analysis is then

performed in Section 3.4, where the advantages and disadvantages of CO-NLS are assessed using an extensive benchmark suite. Section 3.5 concludes this chapter by summarizing and discussing the presented content.

Chapter 4 presents the work performed for complexity control in symbolic regression and motivates the benefit of simpler symbolic regression models. In Section 4.1 existing complexity measures for symbolic regression methods are reviewed and the recursive complexity is defined. In Section 4.2 the advantages of multi-objective symbolic regression are motivated and the applicability of NSGA-II is evaluated. Based on the observations during the evaluation, NSGA-II is slightly adapted to increase its performance when solving multi-objective symbolic regression problems. The performance of NSGA-II with varying complexity measures is compared to standard genetic programming in Section 4.3. Artificial and real-world benchmark problems are used in the performed experiments, where the accuracy as well as the simplicity of generated models is evaluated. Afterwards, the content of this chapter are summarized in Section 4.4.

Finally, the whole thesis is concluded by Chapter 5, which highlights the main research contributions and gives an outlook for future research on the topics discussed in the thesis.

Chapter 2

Symbolic Regression

Symbolic regression is the task of finding a model that describes the relationship between a dependent variable and several independent variables as accurately as possible. This definition applies in general for all regression methods. However, the characteristics that make symbolic regression stand out compared to other regression techniques are that no assumption about the model structure is made a-priori and that the model is generated in symbolic form as an analytical mathematical expression [Koza, 1992]. Hence, symbolic regression is primarily used for function discovery or system identification [Schmidt and Lipson, 2008].

When performing symbolic regression the appropriate model structure, an analytical mathematical expression, to describe the data has to be discovered and simultaneously the parameters for a concrete model structure have to be identified. On the other hand the greater part of regression methods work with a predefined fixed model structure and optimize the parameters of this structure to fit the data. As there are no restrictions and assumptions about the model structure imposed for symbolic regression, the search space of possible solutions to the regression problem is much larger compared to other regression methods. In fact the search space when performing symbolic regression is infinitely large. Due to the finite amount of data used to train the models, there is an arbitrary number of symbolic models describing the data equally well.

Therefore, symbolic regression is generally slower compared to other regression techniques, where only the model parameters of a predefined structure have to be adapted to the given data. Furthermore, due to the size of the search space, heuristic algorithms, which cannot guarantee a certain result in contrast to deterministic algorithms, are generally used to solve symbolic regression problems.

The largest benefit of symbolic regression is that the generated models are present in symbolic form and thus are easily accessible to the user. The symbolic form enables the user to inspect and interpret the models with the goal of gaining knowledge about the underlying system generating the data. Furthermore, the models can be simplified by performing mathematical transformation, their behavior can be analyzed by calculating the according gradient and they can be easily incorporated in other software. However, the benefit of the symbolic form is weakened the more complex a model is. Although a symbolic model compromising of several thousand terms and operands is theoretically still open for interpretation and could be studied in detail, the necessary effort to do so is drastically increased and sometimes interpretation is virtually impossible. Therefore, restrictions on the model complexity are commonly introduced when performing symbolic regression to limit and shrink the search space and to ensure the interpretability of the models.

Another benefit is that symbolic regression exhibits no bias towards specific functions occurring in the models and no domain knowledge about the analyzed data has to be given. For example, when using linear regression to generate a symbolic model, nonlinear relations cannot be learned. A possibility for creating nonlinear symbolic models is to specify the model structure and tune the model parameters accordingly to the data. However, the structure specification has to be performed manually according to the given data and is thereby affected by human bias. On the contrary, symbolic regression does not assume a specific model structure and is therefore not biased towards particular functions, but rather utilizes the most promising ones to model the data. Consequently, symbolic regression can generate unexpected and surprising results, which in turn lead to a deeper understanding about the modeled data.

2.1 Genetic Programming

Symbolic regression has first been introduced by John R. Koza in his book 'Genetic Programming: On the Programming of Computers by Means of Natural Selection' [Koza, 1992], which is one of the reasons, despite the enormous search space, why symbolic regression problems are commonly solved by genetic programming or other heuristic optimization methods. In this section an overview about genetic programming for symbolic regression will be given.

2.1.1 Introduction

Genetic programming has been developed as an evolutionary algorithm that searches for programs that solve a given problem or defined task. The main motivation for the development of genetic programming was to teach computers a general way to solve problems without explicitly programming them. Instead of programming them, examples are provided to the algorithm that should be solved by the generated program.

Genetic programming has been inspired by genetic algorithms [Holland, 1992] and still shares lots of commonalities with them. The main difference is that while genetic algorithms have been developed to work with a fixed length encoding (initially a bit-string), genetic programming uses a variable-length encoding that enables solving problems where the length of a solution is variable. In the first definition of genetic programming programs were encoded as abstract syntax trees that can be easily converted in LISP S-expressions for evaluating the program outcome. Since then other variants of representing programs have been developed, such as stack-based programs that are evaluated on virtual machines [Perkis, 1994], direct acyclic graphs (Cartesian genetic programming) [Miller and Thomson, 2000], binary vectors that are mapped to an arbitrary Backus-Naur form grammar (grammatical evolution) [O'Neill and Ryan, 2012], linear instructions (linear genetic programming) [Brameier, 2005], or as a character string representing individual genes (gene expression programming) [Ferreira, 2006].

Genetic programming, similar to other evolutionary algorithms, works by iteratively creating new solutions from existing ones until a defined stopping criteria is reached. It uses a pool of solutions, the population, which is initially built from randomly created solutions. These solutions are evaluated on the given problem and their performance is expressed as a numerical value termed fitness (objective value). Based on their fitness value solutions are chosen to reproduce (parent selection), where it is more likely for solutions with a high fitness to be selected by introducing a selection bias. The re-

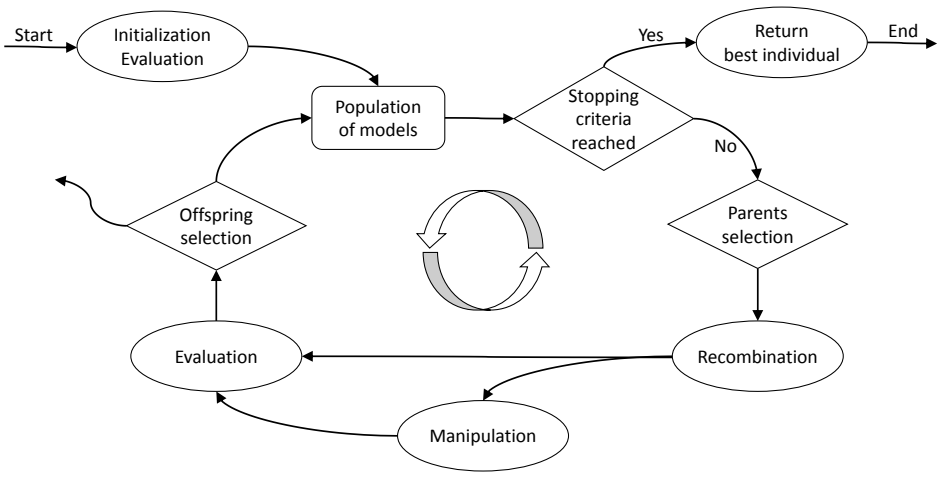


Figure 2.1: Schematic illustration of the genetic programming algorithm.

production works by combining two or more selected solutions to form a new solution. A common method for reproduction is crossover, where parts of the genotype of selected parents is exchanged. Afterwards some of the new solutions are randomly manipulated to introduce new genetic material in the population and to enable the formation of previously unreachable solutions. The newly created solutions are then evaluated on the provided examples and their fitness is calculated. The last step of one iteration of genetic programming is replacing the existing population (or parts of it) with new solutions that again are chosen based on their fitness value (offspring selection). This cycle of parent selection, recombination and manipulation, evaluation, offspring selection and replacement is repeated until the stopping criteria is reached (maximum number of evaluated solutions or a desired objective value is reached) and the best solution found so far is returned as result. A schematic representation of genetic programming is depicted in Figure 2.1.

The main driving forces of population-based and evolutionary algorithms like genetic programming are fitness-based selection and a recombination step that creates new solutions that share some commonalities with their parents. This behavior mimics natural evolution and high quality solutions should evolve over time, as long as necessary building blocks to build high quality solutions are present in the population. This has been proven for genetic algorithms that use a binary encoding for solution representation [Goldberg et al., 1989; Holland, 1992], where short fragments of high quality, so called building blocks, accumulate in solution and at least partially for genetic programming as well [Poli and Langdon, 1998a].

The solution representation, how programs are represented, genetic programming works with, is referred to as genotype, whereas the semantics, what a program does when executed, is called phenotype. Because of the variable-length of the genotype and the possibility to express the same phenotype in multiple ways, the mapping between genotypes and phenotypes is not bijective and different genotypes could be translated to the same phenotype. The program represented by the phenotype is evaluated on the provided examples, where again different phenotypes could lead to the same behavior with respect to the provided examples. As the search for programs is performed in the genotype space, the mapping between the genotype, phenotype and associated program introduces neutrality, where syntactically different programs express exactly the same behavior, and makes the search for good programs more difficult.

Another difficulty genetic programming is affected by is bloat [Poli, 2003], an increase of the genotype without an accompanying increase in fitness. Basically the individuals get larger and hence more complicated without increasing their quality. One reason for bloat is the neutrality of the search space and another is the destructiveness of recombination operations. While recombining parent individuals to form a new individual, important and beneficial building blocks can be destroyed and a larger genotype makes those occurrences less likely. However, more compact genotypes, hence solutions to the optimization problem, are more desirable as these allow an easier interpretation and a faster execution of the programs. One of the simplest controls to counter the phenomenon of bloat is to impose static size limits on the genotype [Poli et al., 2008], which also reduces the search space for possible solutions, but at least partially counters the benefits of a variable length encoding.

2.1.2 Tree-based Genetic Programming

The first developed and still most common encoding for genetic programming individuals are symbolic expression trees; hence the name tree-based genetic programming. Tree structures have the benefit that they are easily extendable and therefore, fit the purpose of genetic programming needing a variable length encoding quite well. A symbolic expression tree consists of internal and leaf nodes, where every leaf node represents a self-contained function or data object and every internal node represents a function with its subtrees as arguments. For example a symbolic expression tree encoding a logic gate is depicted in Figure 2.2. The leave nodes are binary inputs $A1$, $A2$, and $B1$ and the internal nodes logical functions.

The main principles of genetic programming are independent of the en-

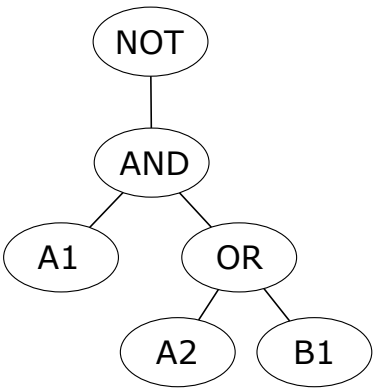


Figure 2.2: Example of a symbolic expression tree encoding a logic gate.

coding used to describe solutions of the optimization problem such as fitness-based parent selection, iterations until a stopping criteria is met, or that newly created child individuals should combine aspects of all parents. However, all the concrete operations for creating, manipulating and evaluating a solution are indeed specific to the used encoding and these operations for tree-based genetic programming are described in the following.

Initialization

Genetic programming starts in general with a population of randomly created individuals. Although, techniques for adapting the initial population to the problem at hand exists (population seeding) and could improve the algorithm's performance, these are only suitable if prior knowledge about the problem exists.

The first methods for random tree initialization have been the *full* and *grow* tree creation methods [Koza, 1992]. Both methods start at a given root node and extend it by adding child nodes until a complete tree is formed. The full method chooses only function nodes while a predetermined maximum tree depth is not reached. When the maximum tree depth is reached only terminal nodes are inserted as child nodes and the tree is completed. As a result all terminal nodes are located on the same tree level and a tree as depicted in Figure 2.2 could not have been created with the full method, because the terminal node 'A1' is not on the same level with the other terminal nodes 'A2' and 'B1'. Another consequence of the full method is that the depth of every created tree is always equal to the maximum tree depth.

The grow method works similarly to the full method with the difference that whenever a new child has to be added, the new node is chosen randomly among all available nodes including terminal nodes. This is the main

difference to the full method that does not allow the insertion of terminal nodes until the maximum tree depth is reached. Therefore, it is possible with the grow method to generate unbalanced trees with terminal nodes in varying levels and the resulting tree does not necessary reach the maximum tree depth at all.

Trees created by the full method are in general larger compared to trees created by the grow method, but also structurally more similar to each other. As none of the two methods for tree creation is regarded better than the other and both methods only require the maximum tree depth as a parameter the so-called *ramped half and half* method for tree creation has been implemented. When using the ramped half and half method 50% of the initial population is created with the full method and the other 50% with the grow method yielding to a more diverse initial population.

The disadvantages of the grow and full method, hence also of the ramped half and half, are that they allow no fine grained control of the particularly created trees with respect to the tree size and frequency of the occurring symbols. The size of the created trees is only bound by the maximum tree depth and all symbols are chosen with equal likelihood. These short comings are addressed by the probabilistic tree creators PTC1 and PTC2 [Luke, 2000a]. Both methods allow the definition of an expected tree size, which should be achieved while creating a new tree. Additionally, PTC2 enables the definition of symbol frequencies for internal and leaf nodes that are respected during tree creation. PTC1 and PTC2 work by maintaining a look-ahead queue of boundary positions at which the tree can be further extended until the desired tree size is reached.

The choice of the tree creator has a strong effect on the size distribution of the created trees. Figure 2.3 highlights this influence by showing the size histogram for the different tree creators when 1000 random trees are created. The maximum tree depth has been restricted to 8 and there have been four binary function symbols and two terminal symbols allowed. The results for the full method are omitted, because the full method always creates trees with a size of $2^8 = 256$ nodes, the maximum possible size for binary trees with a depth restriction of 8. The grow tree creator builds lots of small trees and did not produce any trees containing more than 220 nodes. The behaviors and size distributions of the full and grow tree creators are reflected by the results of the ramped half and half method, where one half of the trees are allocated to the last bin (those created by the full method) and the other half follows the distribution of the grow method. The trees produced by the PTC2 have been created using a uniform distribution $U[1, 256]$ for the expected tree size, which is clearly visible in Figure 2.3.

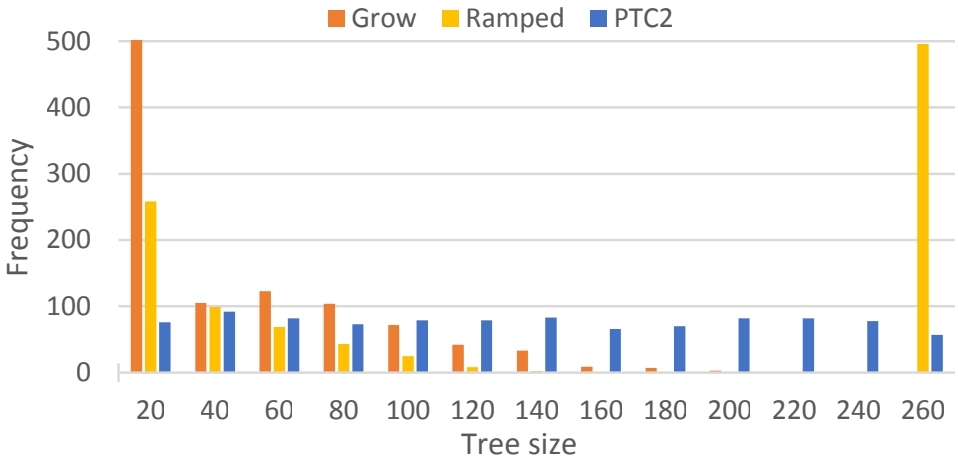


Figure 2.3: Size histogram of tree creators building 1000 random trees with a maximum tree depth of 8.

The reason why the choice of the tree creator in genetic programming is crucial, is that genetic programming depends on the genetic material of the initial population. Therefore, the tree creator highly affects the individuals that can be built by recombination and manipulation during the algorithm execution. A detailed comparison of the effects of the described tree initialization algorithms is performed in Luke and Panait [2001].

Reproduction

In most evolutionary algorithms, recombination and manipulation are responsible for generating new individuals from existing ones and thus navigating through the search space of possible solutions. A prerequisite is the probabilistic selection of parent individuals that are allowed to reproduce. Preferably, individuals with a high fitness are selected for recombination and manipulation, because these should generate fitter individuals.

Selection in genetic programming works similar to genetic algorithms and is based on the fitness of the individuals; in general the higher the fitness value the more likely it is for an individual to be selected as parent. The most common methods for parent selection are *ranking selection* [Baker, 1985], *tournament selection* [Brindle, 1980; Poli et al., 2008] and *fitness proportionate selection* [DeJong, 1975]. More specialized selection methods for genetic programming include *no same mates selection* [Gustafson et al., 2005] or *sexual selection* [Wagner and Affenzeller, 2005]. The effects of using different selection schemes in evolutionary algorithms are highlighted in Goldberg and Deb [1991]; Blickle and Thiele [1996]. Selection mechanisms are in general

independent of genetic programming encodings and can be easily adapted from other evolutionary algorithms as long as they are only dependent on the fitness value. A detailed study on selection mechanisms and their effects on the search behavior of genetic programming was performed in Xie [2009].

After the selection of parent individuals for reproduction, new individuals forming the next generation are created. There are three different ways to generate individuals from existing ones:

- Copying
- Recombination
- Manipulation

Depending on the genetic programming variant used, the three reproduction operations are either used mutually exclusive or occur in combination, for example copying or recombination in conjunction with manipulation. Generally, every reproduction operation is stochastically applied according to a given probability during the algorithm execution. However, it is also possible to specify the number of new individuals that have to be created for each operation, which is especially important for copying.

The simplest reproduction operation is building an exact copy / clone of the selected parent. Copying is used, if at all, on a small portion of the population and the main motivation is to maintain genetic material in the population. Another reason for copying is that elites, the best individuals in the current population, are passed to the next generation without any modifications so that a steady fitness increase without any decline is guaranteed.

Another reproduction operation is the recombination of several parent individuals to form new offspring. The newly created offspring should differ from its parents while still containing parts of their genetic material. The most common form of recombination is crossover, which is an encoding specific operation. Therefore, *subtree crossover* [Koza, 1992] is used in tree-based genetic programming. Subtree crossover works by selecting one or multiple random crossover points in each of the parent individuals and swapping the subtrees beneath the selected crossover points, which is illustrated in Figure 2.4. In its original form crossover points are chosen uniformly and due to the prevalence of crossover points with only a single terminal node below them, terminal nodes as crossover points are more often selected. To counteract this phenomena it is common to determine first whether terminal and function nodes should be selected before choosing the concrete crossover point.

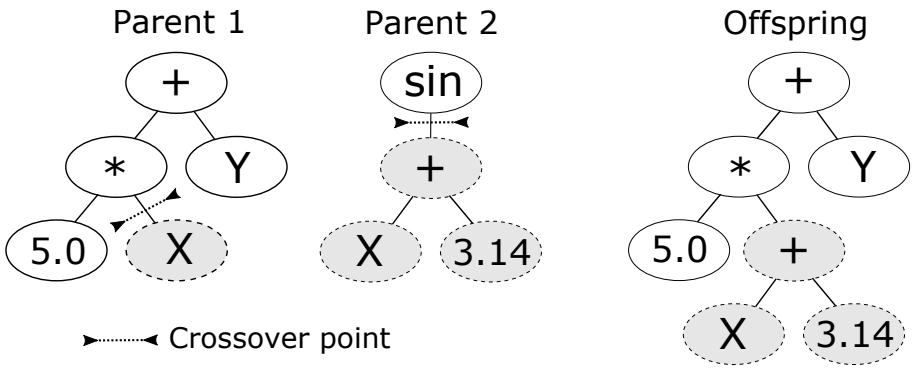


Figure 2.4: Genetic programming crossover exchanging the highlighted subtrees in the parents to form a new offspring individual.

All crossover operations exchange subtrees between the parent individuals. However, more advanced crossovers choose the crossover points, and therefore the exchanged subtrees, more sophisticatedly. For example, the *one-point crossover* for symbolic expression trees [Poli and Langdon, 1998a] chooses the crossover point in one parent, aligns the tree structures and determines the corresponding crossover point to the one chosen in the second parent, therefore maintaining the position of the exchanged genetic material. Other crossovers choose the crossover points according to the size of the exchanged subtrees [Langdon, 2000; Harries and Smith, 1997] or the semantics of the exchanged subtrees [Majeed and Ryan, 2006; Nguyen et al., 2009; Uy et al., 2010]. Although several crossovers have been proposed in recent years and their effects on the search algorithm have been studied [Poli and Langdon, 1998b; Kronberger et al., 2009], the standard *subtree crossover* is most commonly used, presumably due to its simplicity.

The last reproduction operation is manipulation, which is either applied to a copied individual or the resulting individual of a recombination operation. The main purpose of manipulation is the modification of existing individuals and the introduction of new genetic material. Without manipulation, only variations and combinations of genetic material in the initial population can be created. As a result, the space of possible solution that can be reached by genetic programming would be completely dependent on and limited by the genetic material present in the initial population.

The most common form of manipulation is mutation that is responsible for randomly altering the generated individuals. A simple way of introducing new genetic material is to replace an existing subtree with an randomly created one. This operation is called *subtree mutation* or *headless chicken crossover* [Angeline, 1997], because the same effect could be achieved by

performing a subtree crossover with the selected individual and newly created one. Other mutation operations range from *point mutation*, where a single, specific node is changed, *permutation mutation*, which alters the argument order of a node (the order of its subtrees) [Koza, 1992] to *hoist mutation* [Kinnear Jr., 1993] that sets the root of the tree to a random node and thereby creates a new smaller individual. The number of different possible mutation operations seems to be unlimited, yet *subtree* and *point mutation* are prevalent.

Despite mutation that introduces new genetic material in the population, other manipulation operations exist. Simplification is an operation that changes the genotype of an individual without affecting the phenotype. Simplification is termed *editing* in Koza [1992] and the most general editing rule is that any function without side effects, no context dependency and only constants as arguments is replaced with its constant evaluation results; for example the expression $(3+2)$ will be replaced by (5). More specific rules for simplification can be implementing if domain knowledge about the expression semantics is available. For example, the boolean expression (TRUE AND X) gets replaced with (X).

Other common manipulation operations are pruning and encapsulation. Pruning removes or replaces arbitrary subtrees from an individual to reduce the genotype size and counter the phenomena of bloat. Encapsulation is extracting the functionality encoded by a certain tree structure in a separate function so that it can be reused from multiple locations. This can be implemented either by generating automatically defined functions [Koza, 1994], or tagging substructures for reuse [Angeline and Pollack, 1993], or saving partial program evaluation in an additional memory segment [Teller, 1994].

Canonical Genetic Programming

After a rough overview of genetic programming and an explanation of its building blocks, in this section a simple version of the algorithm is described. The genetic programming algorithm is stated in Algorithm 1. Two essential parameters are the population size *PopSize* and the maximum generations Gen_{max} that should be performed, because these parameters predefine how many solutions are evaluated in total.

First the generations counter g is initialized with zero and the first population is filled with randomly created individuals. Afterwards the fitness of the individuals is evaluated. How this is performed is dependent on the problem that should be solved and the objective values, which should get optimized. Next the iterative part of the algorithm starts, which is executed until the generations counter reaches the maximum generations. A new empty popu-

lation is created and until this new population is not completely filled, new individuals are created. This works by selecting parent individuals and generating an offspring individual by copying, recombination, mutation, or a combination thereof. Then the fitness of the offspring is evaluated and it is added to the new population. When the new population is filled, is replaces the existing population and the generations counter is increased. This loop is repeated until the maximum number of generations is reached and the best solution found so for is returned as the result of the algorithm.

This description provides an overview of the necessary steps for genetic programming. In practice several modifications and additional details to this algorithm exists, but the described parts are in one way or another executed for each genetic programming variant.

Algorithm 1 Canonical Genetic Programming

Require: Population size $PopSize$, Maximum generations Gen_{max}
  $g \leftarrow 0$
  $Population_0 \leftarrow$ Create $PopSize$ Individuals
  Evaluate fitness of $Population_0$
  while $g < Gen_{max}$ **do**
    $Population_{g+1} \leftarrow \{\}$
    while $|Population_{g+1}| < PopSize$ **do**
      Select parent individuals from $Population_g$
      Generate *Offspring* from selected parents
      Evaluate fitness of *Offspring*
      $Population_{g+1} \leftarrow Population_{g+1} \cup$ *Offspring*
    end while
    $g \leftarrow g+1$
  end while

Grammar-based Genetic Programming

In the initial definition of genetic programming all function and terminal symbols had to fulfill the closure property [Koza, 1992] that guarantees type consistency and evaluation safety. The closure property requires that every function can handle any function or terminal symbol as its input argument and is well defined regardless of its arguments. The easiest way to ensure type safety is to work with only one representation internally. For example, when logical expressions should be generated, only binary inputs and boolean operations are used. Another simple implementation of type safety is to define automatic conversions between data types such as converting a floating

point number x to a boolean input by a function $f : \Re \rightarrow [T, F]$ that returns T when $x > 0$ and F otherwise, which is automatically applied when a type mismatch is encountered.

The second requirement of the closure property evaluation safety ensures that every function returns a valid evaluation result. This is necessary because functions could fail during program execution such as a division by zero. A way to handle evaluation safety is to redefine the semantics of potentially faulty functions either by ignoring their result or by returning a different value. In the case of a division by zero $x/0$ the inverse of the numerator $1/x$ is returned instead of an error [Koza, 1992].

The type and evaluation limitations of the closure property are overcome by defining types for each function and terminal nodes and in turn imposing limitations on the validity of trees. Functions can only be nested if the return type of its arguments match the necessary argument type of itself. Such a system has been developed under the name of strongly typed genetic programming [Montana, 1995]. Resulting by the restrictions on possibly valid trees, all genetic operations that manipulate expression trees such as creation, crossover, mutation, and editing have to be adapted to take these restrictions into account. While in the standard implementation of the subtree crossover two randomly selected subtrees are exchanged, this is no longer possible, because the choice of the first crossover point in the receiving parent restricts the possible crossover points in the donating parent due to the incompatibility of crossover points with different return types.

Another possibility to impose constraints on allowed expression trees is by extending the concept of strongly typed genetic programming to a more general one, where for every argument of a function and the set of allowed functions and terminals is specified. As an example the unary square root function *sqrt* can be restricted in a way that its argument has to be a function *abs* that calculates the absolute value of its argument. Therefore, evaluation safety is automatically ensured by guaranteeing a positive argument value for the *sqrt* function.

This concept of imposing constraints on allowed expression trees can be further extended and generalized by using grammars, which explicitly define the space of valid expression trees. An extensive survey on the different approaches to grammar guided genetic programming is given in Mckay et al. [2010]. Eventually, the benefits of grammars for guiding the search process and the simple inclusion of semantics in the generated programs led to the separate search algorithm *Grammatical Evolution* [O'Neill and Ryan, 2012].

2.1.3 Symbolic Regression with Genetic Programming

As stated before symbolic regression is a regression analysis method that models the relationship between one dependent and several independent variables in form of a mathematical expression without making any a-prior assumptions about the model structure. When performing genetic programming to solve symbolic regression problems the search for programs that solve a given task becomes the search for mathematical expressions that describe the data most accurately.

Changing a genetic programming system to generate mathematical expressions is achieved by choosing an appropriate function and terminal set. In case of symbolic regression the function set contains mathematical operations, such as arithmetic, trigonometric, or power functions, but in principle any mathematical operation is suitable. The terminal set consists of constant numerical values and variables. An illustrative example is the function set $\mathcal{F} = \{+, -, *\}$ and terminal set $\mathcal{T} = \{x, 1.0\}$ that allows the construction of any polynomial of x with integer coefficients. The explanation is that by recursive application of addition and subtraction of the numeric value 1.0 any integer and by multiplication of x with itself any power of x can be created. For instance the polynomial $3x^2 + x + 1$ can be represented as $(1 + 1 + 1) * (x * x) + x + 1$ which can be constructed from the previously defined function and terminal set.

Besides the function set, the minimum and maximum arity of each function in the set has to be defined. For most cases unary and binary operations are used and the minimum and maximum arity is the same. However, for associative operations a higher maximum arity allows the formation of denser and easier to interpret trees, representing the same expression. Keeping the previous polynomial example $3x^2+x+1$, if only binary functions are allowed, the representation would be $(((1 + 1) + 1) * (x * x)) + x) + 1$ as shown in Figure 2.5 on the left side. The extension of addition and multiplication to allow more subtrees, enables a more compact representation $(1+1+1)*(x*x)+x+1$ indicated in Figure 2.5 on the right side. Exactly the same representation of the polynomial as expression tree cannot be achieved, because the power or square function is not included in the function set $\mathcal{F}$ and the terminal set $\mathcal{T}$ does not include the numeric constant 3.0.

All three representation of the example polynomial are semantically equivalent and can be transformed into each other by applying associative transformation. However, their syntax is different (tree sizes in Figure 2.5) and as genetic programming operates in general on the syntax of the symbolic expression tree, these three representations are not treated as equal (see also genotype-phenotype mapping in Section 2.1.1).

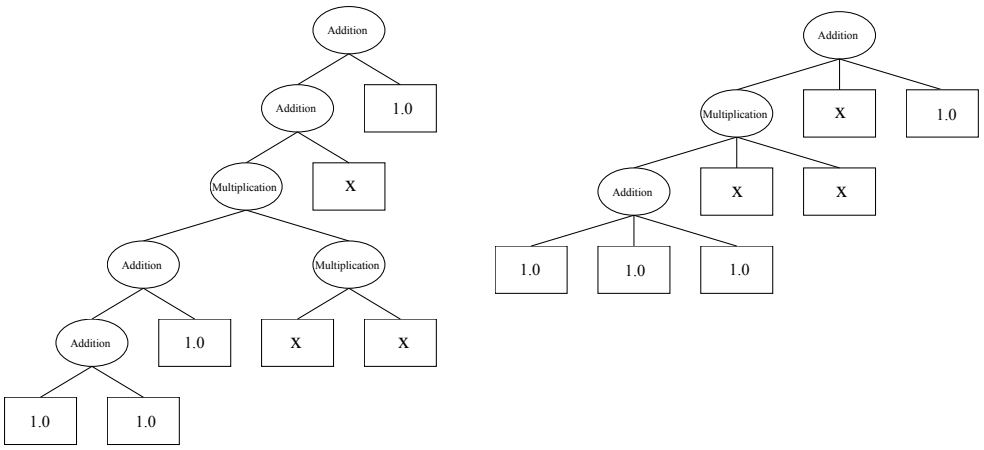


Figure 2.5: Symbolic expression tree representations of the polynomial $3x^2 + x + 1$ in binary and compacted form.

Solution evaluation

Symbolic regression can be reformulated as an optimization problem, where the model which minimizes the error between its estimations and the desired target variable has to be found. Genetic programming is used for navigating through the search space of possible solutions by applying the previously discussed operations for tree creation, parent selection, recombination, and manipulation. The final missing part to solve symbolic regression problems with genetic programming is the solution evaluation, which assigns a fitness (objective) value to a solution candidate. Solution evaluation always depends on the encoding used in genetic programming and the problem which should be solved; in this case tree-based genetic programming solving a symbolic regression problem.

Solution evaluation consists of two consecutive steps, program interpretation and objective value calculation. The interpreted program in combination with provided input data calculates the program result for specific test cases. In the case of symbolic regression, the program is the mathematical expression and the input data are the concrete variable values occurring in the mathematical expression. The interpretation results in a numerical value. How the program interpretation works in detail depends on the genetic programming implementation, but commonly the symbolic expression tree is iterated from the leaf node to its root and the variable values are aggregated according to the encountered functions during this tree traversal. A detailed description of how symbolic expression trees are interpreted in the heuristic optimization framework HeuristicLab is given in Kommenda et al. [2012].

The program interpretation is repeated for every supplied test case yield-

ing a vector of estimated values y', that describes the semantics of the program for the given input data x. As previously stated, the deviation between the actual target values y and the estimated values y' of the solution candidate should be minimized identically to other regression methods. In evolutionary computation, the function mapping the program semantics to a single value is called fitness function. For symbolic regression it is closely related to cost functions in machine learning, with the difference that fitness is maximized, whereas costs are minimized. However, a cost function can easily be turned into a fitness function, for example by taking the inverse value, and vice versa. Therefore, the term 'objective' or 'objective function' is used for the remainder, which should either be minimized or maximized.

Common objective functions in symbolic regression are the mean absolute error (MAE) or the mean squared error (MSE) between the estimated values and the target values. Other often used object functions are the root mean squared error (RMSE), the normalized mean squared error (NMSE), also termed fraction of variance unexplained, and the coefficient of determination (R^2).

2.2 Genetic Programming and Symbolic Regression Software

Although the basic principles of evolutionary algorithms and genetic programming are easily described, the details and necessary operations for a feature complete and efficient genetic programming algorithm solving symbolic regression problems can be hard to implement correctly. A framework can simplify this work tremendously and provides additional functionality for the user, so that the focus can be set on new algorithmic advances and methods. Furthermore, lots of frameworks are shipped with standard algorithms and operations to which new developments can be compared to. The choice of the right framework for a given task depends on the user's proficiency with the used programming language, the already gained experience with the concrete framework, the provided functionality on which new developments can be built upon, as well as available documentation and tutorials to ease its usage. In general, there is no single best framework, only the best framework for the task at hand and for the person performing this task.

In the following, a short overview of existing, actively developed frameworks and programs that support symbolic regression by genetic programming is given. This represents by no means a complete compilation of genetic programming frameworks, but the most frequently used and still actively de-

veloped and maintained programs in the author's opinion are described. A more general and comprehensive list of genetic programming frameworks can be found at the genetic programming homepage[1].

ECJ

ECJ[2][Luke, 2002] is a Java open-source and general-purpose framework for evolutionary computation developed at George Mason University's Evolutionary Computation Laboratory ECLab[3]. It is one of the most widely used [White, 2012; Parejo et al., 2012] and longest available yet actively developed frameworks in evolutionary computation research. *ECJ*'s development has been inspired by the classic genetic programming system *lil-gp* [Zongker and Punch, 1998] that is directly based on the work by Koza [1992]. *lil-gp* is hard to extend and customize which has been the major reason for the development of *ECJ*. Another reason for the development was to have an industrial-grade framework for evolutionary computation and not just genetic programming available that provides useful functionality over a longer period of time.

ECJ provides tree-based genetic programming out of the box with predefined problems such as boolean multiplexer and parity or symbolic regression. It has been widely used for genetic programing research for example on tree creation [Luke and Panait, 2001] or parsimony pressure [Luke, 2002]. Additionally, a reference implementation of well-defined genetic programming and symbolic regression benchmark problems [White et al., 2013] is maintained in *ECJ*.

DEAP

DEAP[4] [Fortin et al., 2012] is the abbreviation for distributed evolutionary algorithms in python and has been developed to test new ideas and for rapid prototyping in the context of evolutionary computation. It is available as open source framework in Python and is maintained by Computer Vision and Systems Laboratory at Université Laval in Quebec, the same research group that is actively developing the C++ framework for evolutionary computation *Open BEAGLE* [Gagne and Parizeau, 2002].

Contrary to other evolutionary computation frameworks, *DEAP* does not necessarily provide ready-to-use algorithms, but rather their essential build-

[1] http://geneticprogramming.com/software/ [Accessed 06-Dec-2016]
[2] http://cs.gmu.edu/~eclab/projects/ecj/ [Accessed 06-Dec-2016]
[3] http://cs.gmu.edu/~eclab/ [Accessed 06-Dec-2016]
[4] https://github.com/DEAP/deap [Accessed 10-Jan-2017]

ing blocks. Evolutionary algorithms and meta-heuristics are then constructed by combining these building blocks. The framework directly supports and eases the algorithm construction and therefore enables the rapid prototyping concept. *DEAP* provides an extensive documentation and tutorials and in combination with the rich Python ecosystem, an easy way to create new algorithms or new variations of existing ones.

Support for genetic programming is provided directly in the form of tree-based genetic programming[5] and utility functions for tree creation, recombination or manipulation, and plotting. Furthermore, *DEAP* supports strongly-typed genetic programming, where for each function its input and output data types are specified and only valid trees according to these type definitions are created. Symbolic regression can be setup similarly to genetic programming[6] and due to the simplicity of *DEAP* it has been used in various research for example on bloat control [Gardner et al., 2011] or dynamic system predictions [Quade et al., 2016].

HeuristicLab

HeuristicLab[7] [Wagner, 2009] is another general purpose framework for evolutionary computation and heuristic optimization methods. It is actively developed by the research group HEAL, located at the University of Applied Sciences Upper Austria since 2002 and available as open-source program since the version *HeuristicLab* 3.3 [Wagner et al., 2014]. *HeuristicLab* is implemented in C# and actively used in teaching, research[8] and industrial projects.

Distinguishing features of *HeuristicLab* are its graphical user interface that allows algorithm configuration, execution, and modeling [Elyasaf and Sipper, 2014] as well as an integrated programming environment for prototyping ideas [Beham et al., 2014]. Furthermore, *HeuristicLab* includes a distributed computing environment that automatically distributes, executes and gathers the results of evolutionary algorithms [Neumüller et al., 2012].

Genetic programming is supported in *HeuristicLab* by an extensive tree-based implementation [Kommenda et al., 2012] that further offers strongly

[5] http://deap.readthedocs.io/en/master/tutorials/advanced/gp.html [Accessed 06-Dec-2016]

[6] http://deap.readthedocs.io/en/master/examples/gp_symbreg.html [Accessed 06-Dec-2016]

[7] http://dev.heuristiclab.com [Accessed 10-Jan-2017]

[8] http://dev.heuristiclab.com/trac.fcgi/wiki/Research [Accessed 10-Jan-2017]

typed symbolic expression trees, grammatical constraints in the form of syntax restrictions, automatically defined functions and a basic implementation of grammatical evolution for comparison purposes. Although, control (e.g., artificial ant or lawn mower problem) or boolean logic problems (e.g., multiplexer or parity problems) are implemented as well, most of the genetic programming functionality is tailored towards symbolic regression and classification problems [Affenzeller et al., 2014].

GPTIPS

GPTIPS[9] [Searson et al., 2010] is a free, open source MATLAB toolbox for performing symbolic regression developed by Dominic Searson. An improved version *GPTIPS 2* [Searson, 2015] is available since May 2015. Using MATLAB as the foundation for the implementation, *GPTIPS* provides an easy to use interface and interactive environment, out of the box multiplatform support, and fast, robust, and trustable matrix and vector operations and algorithms as well as the use of the symbolic engine of MATLAB for post-run analysis and model simplification. Most of the functionality of *GPTIPS* is command line driven and configured by adapting a MATLAB M file that contains the relevant parameters.

Contrary to the other frameworks *GPTIPS* is specifically tailored towards symbolic regression and does not support genetic programming in general. It is a widespread software for performing multi gene genetic programming [Searson, 2002], where a linear combination of multiple expression trees referred to as genes, form a solution to the regression problem at hand. Additional features of *GPTIPS* include steady-state and multi-start [Searson, 2015] genetic programming, various mutation, selection, and termination operations, different complexity measures for expression trees and various reporting and exporting options.

DataModeler

DataModeler[10] is a commercial package extending Wolfram Mathematica for solving symbolic regression problems. Similar to *GPTIPS*, which uses MATLAB internally, *DataModeler* benefits from the powerful functionality, visualization support and symbolic computation engine provided by Wolfram Mathematica.

The main genetic programming paradigm followed in *DataModeler* is ParetoGP [Smits and Kotanchek, 2005] that combines multi-objective op-

[9] https://sites.google.com/site/gptips4matlab/ [Accessed 17-Jan-2017]
[10] http://www.evolved-analytics.com/?q=datamodeler [Accessed 02-Aug-2017]

timization with an additional archive of Pareto-optimal solutions. Due to ParetoGP the problems of bloat and selecting an appropriate model complexity to describe the data, are implicitly reduced, as model accuracy and complexity are equally important optimization objectives. Further features of *DataModeler* are automatic data balancing, extensive model analysis, selection, and life-cycle support, sensitivity analysis and ensembling to build more trustable models [Kotanchek et al., 2008].

Eureqa

Eureqa[11] is a proprietary software for searching analytical models describing provided data. It has been initially developed by the Creative Machines Lab[12] and has subsequently been commercialized by Nutonian. The underlying evolutionary system for generating data-based models is based on research on coevolving fitness predictors [Schmidt and Lipson, 2006, 2008], where multiple subpopulations are used to identify accurate models and subsets of data points that discriminate the models well, which in turn speeds up the optimization significantly. Furthermore, Pareto front exploitation by using age-layered populations [Hornby, 2006; Schmidt and Lipson, 2011] has been investigated for symbolic regression. *Eureqa* gained public attention by a scientific publication [Schmidt and Lipson, 2009], where the laws of conservation of angular momentum of a double pendulum have been automatically discovered by the system. Until the present day a free version of *Eureqa* is available upon request for non-profit academic research.

[11] http://www.nutonian.com/products/eureqa/ [Accessed 02-Aug-2017]
[12] http://www.creativemachineslab.com/eureqa.html [Accessed 02-Aug-2017]

2.3 Deterministic Symbolic Regression

Commonly symbolic regression problems are solved by evolutionary algorithms such as genetic programming or variants of it. A consequence of the heuristic nature of evolutionary algorithm is that the same algorithm applied on the same data yields different results for each execution. This might be beneficial to search enormous solution spaces efficiently, but several advanced machine learning concepts are either rendered useless or significantly hampered in their applicability, such as meta parameter tuning or validation concepts. Furthermore, a technique yielding different results with every execution is less trustable and researchers always have to perform multiple repetitions of the algorithms to report statistical results on obtained solutions, to rule out the possibilities of lucky outliers.

Few deterministic methods have been developed for solving symbolic regression to diminish the disadvantages of heuristic methods. Most of the deterministic symbolic regression algorithms reduce the space of solutions by limiting possible model structures. For example, if the search space is restricted to only contain linear combinations of the input variables, a deterministic algorithm would be to perform ordinary least squares regression for every possible input variable combination and returning the best found model. This algorithm would create 2^N linear models, where N is the number of input variables, and solve the problem of searching for models containing only linear parameters deterministically. The constraint enabling this rather primitive algorithm is to only search for linear combination of untransformed input variables as possible models.

Additionally, determinism is supported by a structured way of navigating the search space and hence avoiding duplicate and already evaluated solutions. In this example algorithm it is achieved by creating solutions in a structured way from the most simplistic to the most complex, which contains all input features. Intelligent iteration of possible solutions, limitation of the search space, and integration of other machine learning algorithms are the main components of deterministic algorithms for symbolic regression and a few algorithms are presented in the following.

2.3.1 Fast Function Extraction

Fast Function Extraction (FFX) [McConaghy, 2011] has been introduced by Trent McConaghy in 2011. The main motivation for developing FFX has been to have a symbolic regression tool that is fast, deterministic and can handle many input variables. In turn symbolic regression should become a technology delivering accurate solutions without the necessity for parameter tuning. FFX has been implemented in Python 2.7[13] using SciPy [Jones et al., 2001–], NumPy [Van Der Walt et al., 2011], and scikit-learn [Pedregosa et al., 2011] for performing numerical computations and machine learnining. FFX is available as open source for non-commercial use at the personal webpage of Trent McConaghy[14] or GitHub[15].

The main idea of FFX is to generate several new features on which regularized learning strategies are applied to create multiple regression models. The generation of new features is done by applying several different basis functions B to the original input variables. As a result FFX is not able to generate arbitrary models, but generates models that belong to the class of generalized linear models (GLMs) [McCullagh and Nelder, 1989] displayed in Equation (2.1).

$$f(x) = a_0 + \sum_{i=1}^{|B|} a_i B_i(x) \tag{2.1}$$

There are two different kinds of basis functions used, univariate and bivariate bases. In a first step all exponential bases of x, $\frac{1}{x_i}$, $\frac{1}{\sqrt{x_i}}$, $\sqrt{x_i}$, and x_i itself, are created and added to the set of basis functions. On all of these basis functions the operators $abs(b), max(b,0), min(b,0)$, and $\log_{10}(b)$ are applied and included in the set of basis functions B. Afterwards hinge functions [Friedman, 1991] $max(0, x-t)$ or $max(0, t-x)$, where the threshold t depends on the range of x_i, and squares x_i^2 are added. The last step for the creation of the basis functions B is to add all products of already existing bases. In the implementation of FFX a maximum on bivariate bases (either 4,000 or 8,000 depending on whether fractions are enabled or disabled) is applied to speed up the calculation, where bivariate bases consisting of univariate bases which have a higher influence on the model are preferred. Furthermore, the created bases are automatically filtered to exclude ill behaving functions (evaluations to ∞ or undefined evaluations), functions without any effect (e.g., $abs(x^2)$), and constant bases $(max(x,0)|max(x) < 0)$.

[13] http://www.python.org [Accessed 16-Aug-2017]
[14] http://trent.st/ffx/ [Accessed 16-Aug-2017]
[15] http://github.com/natekupp/ffx [Accessed 16-Aug-2017]

After the creation of the basis functions, the actual models are built using the elastic net model building technique [Zou and Hastie, 2005]. Elastic net is convex combination of ridge regression [Hoerl and Kennard, 1970] and lasso regression [Tibshirani, 1996]. Ridge regression uses the L2-norm for penalizing large coefficients, but is unable to generate sparse models, whereas lasso regression uses to L1-norm and performs shrinkage and variable selection simultaneously. Equation (2.2) shows formulation of the naive elastic net, which has to be minimized for a specific combination of λ_1 and λ_2 to obtain the model coefficients β.

$$L(\alpha, \beta) = ||y - X\beta||_2^2 + \lambda_1||\beta||_1 + \lambda_2||\beta||_2^2 \tag{2.2}$$

FFX uses a mixing parameter p and combines λ_1 and λ_2, so that $\lambda_1 = p\lambda$ and $\lambda_2 = (1 - p)\lambda$. An elastic net fit is performed for different, decreasing values of λ. As a result the training data has to be modelled more accurately and more bases with non-zero coefficients are included in the model until a predefined maximum value is reached. The learning of the coefficient is repeated for different sets of basis functions and different values of λ to create a diverse set of models. These models are afterwards filtered to remove dominated models with respect to the prediction accuracy and number of used features and a Pareto-front of the non-dominated models is returned as a result.

The advantages of FFX are that it combines well-established machine learning techniques to perform symbolic regression and scales well with the number of samples and variables in the dataset. The default configuration of FFX produces accurate models and because FFX is a deterministic algorithm, cross-validation is easily applicable for parameter tuning. A drawback of FFX is that it is only able to learn models in the form of GLMs (see Equation (2.1)), which are specified by the used basis functions. Furthermore, the generated models can become quite large ($\geq$ 100 basis functions) and although the model is expressed as a linear combination of the basis functions, these large models are quite difficult to interpret.

2.3.2 Prioritized Grammar Enumerations

Prioritized Grammar Enumeration (PGE) [Worm and Chiu, 2013] is another technique to perform deterministic symbolic regression implemented in the GO programming language and available at GitHub[16]. PGE reformulates the symbolic regression problem to a grammatical optimization one, where possible sentences defined by a grammar have to be explored. Generated sentences consist of mathematical expressions and terms depending on the used grammar and represent a prediction model when interpreted correctly. An exemplary grammar for building regression models as defined in Worm and Chiu [2013] is shown in Figure 2.6.

$$
\begin{array}{rcl}
\text{Start} & \rightarrow & E \\
E & \rightarrow & E+T \mid E*T \mid T \\
T & \rightarrow & T-N \mid T/N \mid N \\
N & \rightarrow & \sin(E) \mid \cos(E) \mid \tan(E) \mid \\
 & & \log(E) \mid \exp(E) \mid \sqrt{E} \mid L \\
L & \rightarrow & (E) \mid -(E) \mid (E)^{(E)} \mid \text{Term} \\
\text{Term} & \rightarrow & \text{Constant} \mid \text{Variable}
\end{array}
$$

Figure 2.6: Grammar used to create regression models with Prioritized Grammar Enumeration.

PGE starts with a set of minimal sentences (basis functions) such as $c_0 * x_i$, $c_0 + c_1 * x_i$, $\frac{c_0}{x_i}$, and $c_0 * f(x_i)$ and stores these basis functions in a Pareto Priority Queue (PPQ). The PPQ is generated by non-dominate sorting of all generated sentences according to their accuracy and length and pushing the resulting Pareto frontiers to the queue, where each frontier is sorted ascending by length. After the initial generation of basis functions and creation of the priority queue, the first p sentences are popped from the queue and according the expansion rules new sentences are created. The expansion rules applied to sentences are directly related to the production rules of the grammar and include methods for adding new terms, widening, or deepening existing terms. The newly generated sentences are afterwards inserted in the PPQ and the process continues until a defined stopping criteria such as a desired accuracy or execution time is reached. The PPQ balances the trade-off between the exploration of accurate and parsimonious sentences.

[16] http://github.com/verdverm/go-pge [Accessed 18-Aug-2017]

PGE reduces the infinite search spaces of possible sentences by including the semantics of a sentences and reducing them to their canonical form. For example, constant expressions such as $2+4$ or $\sin(\pi)$ are automatically folded and terms of commutative symbols (addition and multiplication) are ordered to achieve the canonical form. Therefore, the search space is structured and reduces isomorphic formulas to their simplest representation. The formulas already evaluated by PGE are kept in a trie structure to avoid reevaluation of equivalent equations, hence saving execution time.

The reduction of the search space is only possible due to omitting of concrete numerical values for constants. Instead of specifying numerical values for generated formulas, a parameter c_i is inserted representing an arbitrary numerical value. Otherwise, formulas with the same structure (operators and variables) but different numerical values, could not be matched in the trie and would be treated as distinct. Afterwards, if the formula has not been previously encountered and therefore is not present in the trie, the numerical parameters c_i are fitted to the training data by using a nonlinear regression method, the Levenberg-Marquards algorithm [Levenberg, 1944]. This approach is similar to the one presented in Section 3.3 and allows the separation of searching for structures from their concrete parameterization, thus narrowing the search space of possible formulas even more.

In summary, PGE achieves deterministic symbolic regression by recursive application of expansion rules on already generated formulas or initially on defined basis functions, the use of a priority queue for steering the search, intelligent structuring of the search space by only considering the canonical representation of the formula, and the separation of the structure and parameters of a formula by using nonlinear regression to find the best numerical values. It contains only two parameters, p to steer the search and the stopping criteria and due to its determinism does not need to be executed multiple times in contrast to heuristic symbolic regression methods. A drawback of PGE is that with an increasing number of features its runtime increases drastically, because the search space grows exponentially with the number of features and the feature selection capabilities of PGE have not been evaluated yet.

Chapter 3

Local Optimization

In contrast to global optimization, where the global optimum should be found, local optimization aims to quickly find a local optimum based on a specific starting point. Hence, local optimization can be interpreted as refinement method for solutions. Memetic algorithms combine such refinement methods with global optimization methods, often population-based, evolutionary algorithms [Chen et al., 2011]. In their initial definition a memetic algorithm [Moscato et al., 1989] is a genetic algorithm [Goldberg et al., 1989] hybridized with hill climbing.

In the context of symbolic regression, local optimization refers to a further improvement of existing solutions towards a local optimum, as for example in Krawiec [2001] or Juárez-Smith and Trujillo [2016]. In the past several attempts to hybridize evolutionary algorithms with machine learning methods have been implemented to achieve more accurate solution to symbolic regression [Raidl, 1998; Lane et al., 2014; La Cava et al., 2015; La Cava and Moore, 2017; Castelli et al., 2015] or classification problems [Wang et al., 2011].

In this chapter the focus lies on local optimization methods for adapting the numerical constants of symbolic regression solutions and how this helps genetic programming to produce more accurate results.

3.1 Constants in Symbolic Regression

3.1.1 Overview

When performing symbolic regression the numerical constant values are of prime importance. Numerical values, referred to as constants, form together with variables the terminal set $\mathcal{T}$. Elements from the terminal set are chosen whenever a leaf node is created in a symbolic expression tree. Constants can be explicitly stated in the terminal set $\mathcal{T}$. For example, the terminal set can be defined as $\mathcal{T} = \{X, 1.0, 2.0, \pi\}$ containing one variable X and the three numerical values 1.0, 2.0, and π. Adding predefined and immutable numerical constants directly to the terminal set has a few implications.

Firstly, the ratio between variables and constants is altered resulting in different probabilities for the terminals to be chosen. In the previous example, it is three times as likely for a constant to be selected during tree creation when a terminal symbol is picked. Hence, the created trees will contain more constants and fewer variables. Whether this poses a problem for solving the symbolic regression at hand depends on the concrete problem. However, the user should be aware of the introduced bias towards selecting constants as leaf nodes. This can be mitigated if the genetic programming algorithm supports symbol frequencies and if these frequencies are adapted according to the number of variables and constants. Symbol frequencies specify a weight according to which possible symbols during tree creation are selected. To achieve a uniform distribution between variables and constants, the symbol frequencies are $\frac{1}{2}$ for X and $\frac{1}{6}$ for the constants. Hence, it is three times as likely for X to be included during tree creation compared to a concrete constant, which compensates the fact that there are three constants and only one variable.

Secondly, due to the immutability of the symbol it is hard or sometimes even impossible to create arbitrary numeric values. Other numeric values, except the predefined ones in the terminal set, can only be created by combining existing constants through the allowed function symbols. In the provided example, the number 4.0 cannot be referenced directly, but can be generated by addition, multiplication or power functions that combine the constant 2.0 with itself. Although there are multiple rather simple ways to generate the number 4.0, 0.2 has to be expressed as $\frac{2}{10}$ and takes several calculation steps (e.g., $\frac{1+1}{((2+2)+1)*2)}$). The more calculation steps are necessary to compute a numeric value the harder it is for the algorithm solving the symbolic regression problem to achieve high quality solutions, because more time is spent on generating the appropriate constants that would be better dedicated to evaluating different model structures and variables.

3.1.2 Ephemeral Random Constants

Instead of adding every numerical constant to the terminal set beforehand, which yields the problems described in the previous section, numerical constants can be introduced by the use of ephemeral random constants ERCs [Koza, 1992]. When ERCs are used, the special symbol $\mathcal{R}$ is added to the terminal set and every time the ERC symbol $\mathcal{R}$ is selected during tree creation a new constant value is drawn from a predefined distribution. Common distributions the numeric values are drawn from are the uniform distribution or the Gaussian distribution. The properties of the distributions, such as the lower and upper limit or the average and mean, for the ERC symbol have to be adapted at the problem at hand to generate appropriate real-valued constants. Once the numeric values are generated, these values remain fixed and are not sampled again. In its initial definition, similarly to the constants added directly as symbols, ERCs once generated are not modified anymore and are moved between the individual solutions by the crossover operator. As a result the constant values can be combined in a way to generate intermediate values necessary for solving the symbolic regression problem. ERCs provide a greater flexibility, because it is possible to create real-values constants according to a predefined distribution, compared to adding constants directly to the terminal set.

A disadvantage of the discussed methods for constant creation is that once created values are immutable. Therefore, the set of potentially reachable constant values through recombination of solutions, depends solely on the initial constant values during tree creation. To mitigate this influence, it is recommended to add special manipulation operators to the algorithm that alter the constant values of a solution. These manipulation operators have been described in Schoenauer et al. [1996], where a random Gaussian variable is added to the constant, which is inspired by mutation in evolution strategies [Schwefel, 1981]. A similar technique by Gaussian mutation is detailed in Ryan and Keijzer [2003] that additionally indicates the problems of finding appropriate numeric constant values. Another possibility inspired by simulated annealing [Kirkpatrick et al., 1983] is to replace all numeric constants with new values, sampled at random from a uniform distribution adapted by a temperature factor [Fernandez and Evett, 1998]. Hence, new numeric values can either be created by the combination of constants through mathematical operations or by manipulation of existing constants. As a result it becomes easier for the algorithm to find the appropriate constant numeric values.

Constant Creation in HeuristicLab

As stated previously, the terminal set for symbolic regression contains constants and variables. If ERCs are used, only one symbol is responsible for handling all numeric values in the generated solutions. In HeuristicLab the same applies to the variable symbol. Instead of having distinct symbols for each different variable, there is one general variable symbol v that ,when chosen during tree creation, is instantiated with one of the allowed variables. Therefore, the terminal set for symbolic regression in HeuristicLab only consists of the ERC symbol and variable symbol; $\mathcal{T} = \{r, v\}$.

Another difference to most other programs for solving symbolic regression problems is that numerical values are not only used in the ERC symbol. In general r represents a numeric constant and v represents a variable. However in HeuristicLab, v is actually a combined symbol that represents a selected feature and additionally contains a weighting factor w. During solution evaluation, the weighting factor is multiplied with the variable value and acts as a scaling term. The weighting factor w is identically generated as ERCs values, but a different sampling distribution can be specified. The combination of the variable symbol with a weighting term can be seen as the more general approach, because weighting can be easily disabled. If w is sampled from a custom distribution that always returns exactly 1.00, the symbol behaves as if no weighting factor would be used, due to the fact that 1.00 is the neutral element for multiplication.

HeuristicLab provides a general concept for manipulating nodes of symbolic expression trees termed *shaking operation*. A *shaking operation* adapts the local parameters of tree nodes. The local parameters of a tree node depend on the symbol represented by the node. Most internal nodes of the expression tree use mathematical operations such as addition or division and do not have any local parameters. In contrast to these, both types of terminals (constants and variables) have local parameters. The numeric value of constants is a local parameter and variables consist of the weighting value and the selected variable itself.

Shaking Operations adapt these local parameters. Numeric values such as the variable weight or the constant values are altered by either addition or multiplication of a value sample from a Gaussian distribution. Additionally, with a given probability a *shaking operation* changes the currently selected variable. Per default, constants are initialized uniformly $\mathcal{U}[-20.0, 20.0]$ and the Gaussian for additive manipulation is $\mathcal{N}(0, 1)$ and for multiplicative manipulation $\mathcal{N}(1, 0.03)$. Variable weights are initialized from a Gaussian $\mathcal{N}(1, 1)$, the Gaussian for additive manipulation is $\mathcal{N}(0, 0.05)$ and for multiplicative manipulation $\mathcal{N}(1, 0.03)$ and the probability to change the selected

variable is 0.2. Whether a numeric constant is manipulated by addition or multiplication is decided randomly with equal probability.

There are two distinct operators that perform *shaking operations* of tree nodes that differ in the number of affected tree nodes. The *one point shaker* adapts one randomly selected tree node with local parameter, whereas the *full tree shaker* adapts every tree node.

3.1.3 Linear Scaling

In the previous section two different ways of integrating numerical constants, by direct addition to the terminal set or by using ERCs, have been discussed. Additionally, possibilities to generate new numeric values based on combination of existing constants or manipulation of the numeric values have been introduced. These are the main concepts for handling numeric values in symbolic regression.

However, in more powerful symbolic regression systems a method called linear scaling [Keijzer, 2003, 2004] is integrated, which also touches constant creation. Instead of directly using the output values of the symbolic expression tree for objective value calculation, the outputs are scaled to the range of the target variable. This changes the search goal from finding the best solution $f(x, w)$ (a function f of the available input variables x and weights w) to finding the best solution that can be scaled linearly $\alpha + \beta * f(x, w)$ and minimizes the objective value. As a result the algorithm solving the symbolic regression problem does not need to find the correct offset and scale for the estimates.

The scaling parameters are dependent on the objective value that should be optimized. For example, if the mean squared error is used to assess solutions, then

$$\beta = \frac{\mathrm{cov}(e, y)}{\mathrm{var}(e)}$$
$$\alpha = \bar{y} - \beta\ \bar{e}$$

where e are the estimated values by the model and y are the target values [Keijzer, 2004]. When using the scaled mean squared error as objective value, the quality of solutions changes and the selection mechanism of genetic programming is highly affected. The scaled mean squared error is bounded by the variance of the target values y [Keijzer, 2004] and therefore the worst possible solution is a constant prediction model that predicts the average of the target values.

An example highlighting the benefits of linear scaling is depicted in Figure 3.1 (adapted from [Keijzer, 2004]). The target values Y that have to be

estimated are calculated by $Y = 0.3x \sin(6.5x)$ and displayed in blue and the estimates of two possible symbolic regression models M_1 and M_2 are plotted as well. M_1 is a constant model always predicting 0.0 while M_2 depends on x. If both models are unscaled (Figure 3.1a) the constant model M_1 has obviously a smaller mean squared error than M_2, mainly because it predicts the mean of Y rather accurately. Nonetheless, M_1 fails to capture the characteristics of Y. If these two models were part of the genetic programming population, the likelihood of M_1 to be selected for reproduction would be increased due to its better fitness.

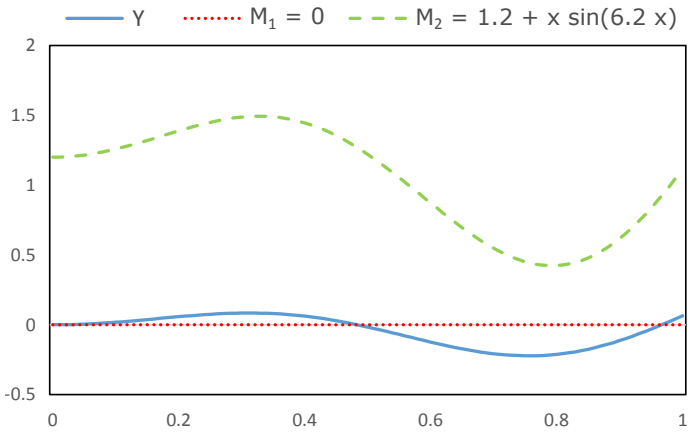
The whole picture changes if both models are linearly scaled to fit the target values Y. Because M_1 does not depend on x, it is altered to estimate the average of Y. The unscaled model M_2 already has a similar shape to the target values Y, although its range of values is different. Linear scaling adjusts M_2 to the same value range and the scaled model achieves a very good fit to the target values (Figure 3.1b). Therefore, in contrast to the unscaled case, the mean squared error reflects the intuitive assumption that M_2 is better suited to model Y than M_1.

An alternative to linear scaling is to use the Pearson's R^2 [Draper et al., 1966], also termed coefficient of determination, as objective value instead of the scaled mean squared error.

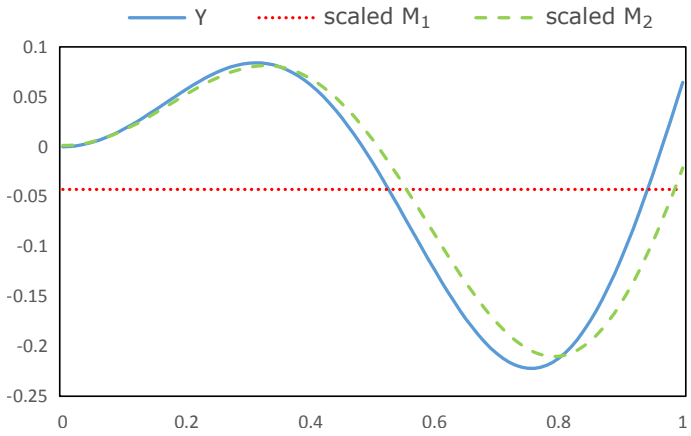
$$R^2(x, y) = \frac{\text{cov}^2(x, y)}{\text{var}(x)\text{var}(y)}$$

When using the Pearson's R^2 the objective value is maximized, whereas if an error measure is used as objective value it will be minimized. The R^2 describes the linear correlation between two variables and ranges from 1.0 for a perfect linear relationship to 0.0 for no dependence between the variables. It is proven that maximization of the R^2 is equivalent to minimizing the scaled mean squared error [Keijzer, 2004], with the exception that the R^2 value is bounded on both sides.

Using linear scaling improves algorithms solving symbolic regression problems significantly, as it removes the necessity to find the optimal scale and offset for the models while only adding a minimal overhead for calculating the scaling factors. As a result it is always recommended to integrate linear scaling when solving symbolic regression problems. However, linear scaling can only adjust linear shifts in the model. This is indicated in the example in Figure 3.1, where the scaled model M_2 does not achieve a perfect fit, because of the wrong numeric constant inside the sine function that cannot be adapted by linear scaling.



(a) Unscaled models



(b) Scaled models

Figure 3.1: Effects of linear scaling of two symbolic models M_1 and M_2 estimating y. While the constant model M_1 has a lower error in the unscaled case (a), M_2 estimates y better when both models are linearly scaled (b).

3.2 Constants Optimization

In the previous section numerical constants in symbolic regression, and the concept of linear scaling are detailed. Following this theme of improving the creation and adaption of numerical constants in symbolic regression solutions, constants optimization is the next logical step. Constants optimization specifically deals with adapting numerical values to increase the accuracy of generated solutions. The term constants optimization originates from the initial definition of symbolic regression, where either a variable or constant symbol is used in the leaf nodes of symbolic expression trees. Although, constants optimization alters the constants of symbolic regression solutions, thus contradicting its definition, the term indicates that the numerical values of a model are adapted. To put constants optimization in the right context and to illustrate its benefits, the main properties of symbolic regression have to be recalled.

The distinguishing characteristics of solving symbolic regression problems are that solutions are learned in the form of mathematical formula, without the necessity of a-priori assumptions about the structure of the model and the necessary input features. As a consequence three interrelated subtasks have to be solved for generating high quality solutions to symbolic regression problems:

1. Selection of the appropriate subset of variables (feature selection)
2. Detection of the best suited model structure containing these variables
3. Determination of optimal numerical constants and weights of the model

Each of these subtask depends on the results of the previous step to generate optimal solutions and therefore improvements of one task can lead to significant improvements to the whole algorithm solving symbolic regression problems. Although, these three subtasks have to be solved in any case, most algorithms create solutions without explicitly addressing the necessary steps. For example, symbolic regression problems are commonly solved by tree-based genetic programming that combines all characteristics of a solution such as the appropriate subset of variables, model structure, and numerical constants and weights in one entity, the individual. Furthermore, individuals are in general manipulated as a whole (by crossover or mutation), which results in additional difficulties for generating good solutions. The reason is that the crossover simultaneously modifies the structure, occurring variables, and weights and only if all those three properties are appropriately adapted, a good solution is discovered.

Constants optimization deals with the last subtask on how to determine optimal numerical constants and weights for a model with a fixed structure and subset of variables. Now instead of handling all three subtasks simultaneously, the parameterization of a model through adapting its numerical constants is separated and solved independently. Hence, genetic programming is responsible for variable selection and model structure generation, whereas determining numeric values is performed by constants optimization. This is a particular appropriate division of tasks, because genetic programming is already well suited for variable selection [Stijven et al., 2011] and could be further improved by integrating established feature selection techniques [Guyon and Elisseeff, 2003; Chen et al., 2017].

Furthermore, generating the best-suited model structure can be reformulated as a combinatorial optimization problem (if restrictions on the model size are applied to limit the solution space). As genetic programming originates from genetic algorithms,which were invented for binary and adapted to combinatorial optimization problems, the suitability of genetic programming for this task is ensured. In contrast to this, genetic algorithms and programming are not the best suited algorithms for real-valued optimization, to which determining optimal numerical constants belongs to. Genetic algorithms and genetic programming are often outperformed by variants of evolution strategies [Schwefel, 1981; Hansen et al., 2003] when performing real-valued optimization.

3.2.1 Related Work

Since the first attempts to solve symbolic regression problems with genetic programming, several methods have been implemented to improve handling of numeric constants in the models. In general, the applied methods can be divided into the following three different research directions:

- Hybridization of genetic programming with other algorithms
- Usage of genetic programming as feature generator for other machine learning algorithms
- Alternative encoding of solutions or reformulation of the optimization goals

Algorithm Hybridization

The most common approach that affects how numerical constants are treated in symbolic regression is the hybridization of genetic programing with other

algorithms. These hybridized algorithms are often heuristic or even evolutionary algorithms as well, possibly due to the attribution of genetic programming to this field, and consequentially the expertise of the researchers with evolutionary algorithms.

One of the first algorithmic hybridization has been GA-P [Howard and D'Angelo, 1995] that combines a linear genetic algorithm with genetic programming for performing symbolic regression. In GA-P the encoding of individuals is split into the symbolic expression representing the model structure manipulated by genetic programming and a binary string that represents the parameterization of this model structure (numeric coefficients) manipulated by a genetic algorithm. Before an individual is evaluated, the constants of the model are replaced by the according part of the binary string and only then the quality of the model can be assessed. This in general increases the runtime of genetic programming by the additional optimization step, but in turn should lead to better symbolic regression solutions.

A similar attempt for constant optimization and creation [Zhang et al., 2007] has been the hybridization of gene expression programming with differential evolution [Storn and Price, 1997]. Differential evolution has also been used in Mukherjee and Eppstein [2012] for co-evolving constants, where it is combined with tree-based genetic programming. Sharman et al. [1995] used genetic programming for the evolution of signal process algorithms that has been enhanced by simulated annealing [Kirkpatrick et al., 1983] to adapt the numeric parameters of the evolved algorithms. The same concept but using evolution strategies [Schwefel, 1981] instead of differential evolution or simulated annealing has been employed by Winkler [2008] or Alonso et al. [2009].

All these methods have in common that meta-heuristics have been applied for tuning the numerical constants of the symbolic expression trees evolved by genetic programming. A difference between the methods is the concrete algorithm used and whether all trees in the genetic programming population or only selected ones are tuned and how the additional effort for constants optimization is distributed.

Integration of Machine Learning Approaches

Another possibility for the optimization of constants is to integrate machine learning approaches. This is especially suitable, because symbolic regression itself belongs to the field of machine learning, where prediction models have to be created. One of the first attempts has been made by Raidl [1998], where genetic programming is combined with multi-variate linear regression [Draper et al., 1966]. There, the separability of the models is exploited by dividing a

model into several parts whenever an addition or subtraction is encountered. These parts are then weighted by numerical factors and because all these parts are linearly combined, linear regression (method of least squares) is applied to determine the individual weights.

A related idea of linearly combining parts of the solutions has been implemented in multiple regression genetic programming (MRGP) [Arnaldo et al., 2014]. A significant difference is that MRGP internally builds all possible subtrees instead of only the ones that are linked by addition and subtraction. As a result much more different parts are created, more weights have to be determined and additionally the individual parts can highly correlate with each other, due to the fact that the created parts are overlapping. Subsequently, instead of applying the method of least-squares for weights determination, least angle regression (LARS) [Efron et al., 2004] is applied that includes constraints on the regression coefficients (weights) and forward feature selection.

Another perspective of the two presented approaches for combining parts of the model is that genetic programming is used as an algorithm for feature construction and the final symbolic regression model is built by least-squares or least angles regression. The benefit of linear model creation is that these linear models are still open for inspection and interpretation. The same concept has also been applied for feature construction for classification tasks [Neshatian et al., 2012] and provides a good overview on the topic. It can be argued that FFX [McConaghy, 2011] can also be regarded as a combination of feature construction, feature selection and parameter optimization method. However, a deterministic basis function creation algorithm (see Section 2.3.1 Fast Function Extraction for further details) is used for feature construction instead of an evolutionary, heuristic algorithm and the models are constructed by multiple linear regression with elastic net regularization.

One of the first attempts of specifically improving the numeric parameters of a genetic programming solution with mathematical optimization algorithms has been performed by Toropov and Alvarez [1998]. This publication introduces genetic programming for generating the model structure for the multipoint approximation method. Its benefits are demonstrated by solving an exemplary problem, where a three-bar truss has to be designed. An new aspect of the publication is that the model generated by genetic programming is not used directly, but simplified and improved before it is reported. Simplification is performed by removing model parts with little or no contribution to the model output, e.g. addition and subtraction of multiple constants. The improvement is performed by taking the original solution and applying a derivative-based optimization method that adapts the numerical parameters so that the sum of squared errors between the model output and the

observation is minimized. However, the advantages and disadvantages of this improvement or model tuning are not discussed in this paper and it remains unclear which derivate-based optimization method has been used in detail.

Gradient-based Optimization

The idea to use gradient-based or derivative-based optimization for tuning the numeric constants of symbolic regression solutions has been picked-up in several studies. Z-Flores et al. [2014, 2015] use a Gauss-Newton method, specifically the Trust Region optimizer [Coleman and Li, 1996], for improving the performance of genetic programming when solving symbolic regression and binary classification problems. A distinction to other constant optimization methods is that for each distinct function of the model a numeric coefficient or parameter is linked to it and the result of the function when evaluated is multiplied by this coefficient. Hence, instead of manipulating the naturally evolved constants of the model, the artificially introduced parameters are adapted. Furthermore, if the model contains for example multiple instances of the same function (e.g., several occurrences of divisions) all these functions share the same coefficient. In the cited research several different ways of integrating this local search have been compared to each other and to standard genetic programming. In detail the number of optimized solutions of the algorithm have been varied, whether only the best, the worst, a specific subset, a randomly chosen, or all solutions per iteration are adapted. The results across six different, representative benchmark problems show that adapting all solutions performs best. Concluding, the local search investigated here, uses a gradient-based optimization method for adapting newly introduced coefficients of the model, but reformulates the overall objective function of genetic programming from generating the most accurate model to generating the model, which can be adapted best to the given data, when additional parameters are introduced.

A similar line of research is followed by Chen et al. [2015], where the generalization aspects of genetic programming with gradient descent for symbolic regression are investigated. In addition to varying the number of individuals that are chosen for constants optimization, the number of steps the gradient descent is applied and at which algorithm iteration gradient descent is performed is varied as well. In total six different configurations for constants optimization, ranging from applying it to every individual each generation to only applying it to the best individual in the last generation, are compared with each other and with standard genetic programming on five noise-free benchmark problems. The algorithm performance in terms of achieved quality and computation time, as well as the generalization capabilities of the

generated solutions on data of a different domain are compared. It is concluded that constants optimization generally improves training performance, but test performance could be decreased due to overfitting the data.

Using gradient-based or derivative-based optimization techniques for constants optimization in symbolic regression applications has been introduced by Topchy and Punch [2001] and this publication builds the foundation for the discussed studies [Z-Flores et al., 2014, 2015; Chen et al., 2015]. There, gradient descent in combination with automatic differentiation (also termed algorithmic or computation differentiation) [Griewank and Walther, 2008] is presented and an increase in training performance is demonstrated on five benchmark problems. The test performance has not been evaluated, however the same algorithm and benchmark problems were used in Chen et al. [2015] that focuses on the generalization aspects. Gradient-based optimization of constants in symbolic regression provides an efficient solution for numeric parameterization of model structures and therefore enables different approaches than genetic programming for solving symbolic regression problems [Worm and Chiu, 2013].

Although the results obtained by including constants optimization in genetic programming show an improvement compared to standard genetic programming, the use of constants optimization is not as wide-spread as one might expect and just in recent years started gaining momentum. Reasons for this might be that constants optimization further complicates the algorithm in terms of finding appropriate parameter settings and implementing it correctly.

Another more compelling reason might be that Keijzer [2003] compared linear scaling directly with the results obtained by genetic programming with gradient descent [Topchy and Punch, 2001] and observed that the improvements of linear scaling considerably outperform the improvements obtained by gradient descent. Admittedly, the performance of the two approaches have not been compared directly, because the underlying genetic programming algorithms differ, but the performance improvements have been evaluated on the same test problems and the base line for comparison has always been calculated for the system at hand. These findings seem counterintuitive, because constants optimization should be a more general approach than linear scaling for optimizing solutions to a symbolic regression problem, but highlight the importance of determining the appropriate scale and offset for the output values of the solutions.

3.3 Constants Optimization by Nonlinear Least Squares

We developed a new approach for constants optimization in symbolic regression that combines the strength of linear scaling with gradient-based optimization techniques [Kommenda et al., 2013a,b]. This new implementation has been that inspired by the improvements obtained by linear scaling. We believe symbolic regression solutions can benefit even more from a general constants optimization methodology. Furthermore, due to the advances in hardware and computing technologies the drawback of spending additional resources for constants optimization is negligible if better solutions for the problem at hand are discovered. This new approach for constants optimization is termed Constants Optimization by Nonlinear Least Squares (CO-NLS) and has been implemented in HeuristicLab [Wagner et al., 2014]. In the previous Section 3.2 the concept of constants optimization and several related techniques have been explained. In the following, the details of CO-NLS are explained and its advantages and disadvantages are shown.

The goal of symbolic regression is to find the model which minimizes the prediction error between its estimates $f(x)$ and target values y. Constants optimization on the other hand tries to optimize the numerical values β of a concrete model $f(x, \beta)$ so that the prediction error of this model is minimal. For example, a symbolic regression model is depicted in Figure 3.2. This model is equivalent to the mathematical formula shown in Equation (3.1), where numerical values are displayed in decimal notation instead of scientific notation and every term encompassed by parentheses represents a terminal node in the symbolic expression tree. Four numerical constants, $\beta = [0.65106, -1.3160, 1.5156, -17.619]$, are present in the symbolic regression model, where two of them act as weighting factors for variables. The question remains whether these concrete values are optimal in a way that the model's prediction error is minimized, or if a better parameterization of the model with respect to β exists.

$$f(x_1, x_2) = \frac{(0.65106 \cdot x_2) - (-1.3160)}{(1.5156 \cdot x_1) - (-17.619)} \tag{3.1}$$

3.3.1 Levenberg-Marquardt Algorithm

The adaptation of the constants in the symbolic regression model is performed by the Levenberg-Marquardt (LM) algorithm [Levenberg, 1944; Marquardt, 1963]. It is a least squares minimization algorithm for models that

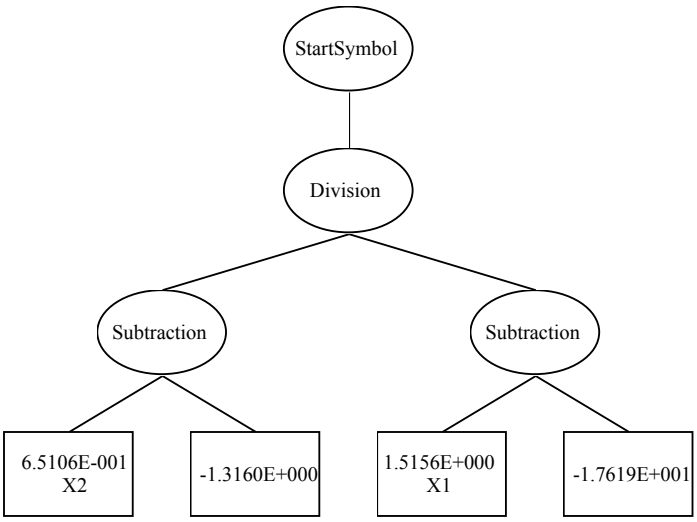


Figure 3.2: Exemplary symbolic regression model

are nonlinear in their parameters (e.g., $f(x) = e^{\beta_1 x}$). When solving symbolic regression problems the candidate models that have to be evaluated are not guaranteed to be linear in their parameters. Hence, nonlinear regression methods, such as the LM algorithm have to be chosen for constants optimization.

The LM algorithm minimizes the sum of squares between a model $f(x, \beta)$ and target values y (Equation (3.2)) over m data points by adapting the parameters β of a model.

$$\underset{\beta}{\operatorname{argmin}} \sum_{i=0}^{m} (y_i - f(x_i, \beta))^2 \tag{3.2}$$

The LM algorithm performs an iterative update of the parameter vector β starting from given initial values. It can be interpreted as a mixture between steepest descent and Gauss-Newton algorithm [Björck, 1996]. Similar to other gradient-based optimization methods the LM algorithm does not guarantee to find the global minimum, but rather converges to the next local minimum depending on the initial starting values.

The update of the parameter vector depends on the gradient with respect to β and a scaling factor δ (Equation (3.4)). As the optimization is performed for multiple data points and parameters the gradient information $\bigtriangledown f$ (Equation (3.3)) for each data point is stored as a matrix commonly referred to as Jacobian. A consequence of using the LM algorithm for constants optimization is that only differentiable functions can be used in the

symbolic expression trees, because otherwise gradient calculation and in turn the LM algorithm would fail.

$$\bigtriangledown f = \left(\frac{\partial f}{\partial \beta_1}, \frac{\partial f}{\partial \beta_2}, ..., \frac{\partial f}{\partial \beta_n} \right) \tag{3.3}$$

$$\beta_{new} = \beta + \delta \cdot \bigtriangledown f \tag{3.4}$$

The iterative update of the parameter vector β is performed until a predefined stopping criteria is reached. The number of calculated iterations or the convergence of the algorithm regarding the gradient norm, the step size, or the change in the function evaluation are commonly used as stopping criteria.

In a recent publication several methods for constants optimization of symbolic regression models have been benchmarked against each other [de Melo et al., 2015]. The compared methods were simulated annealing (SA) [Kirkpatrick et al., 1983], the Broyden-Fletcher-Goldberg-Shannon (BFGS) algorithm [Fletcher, 1987], the conjugate gradient (CG) method [Hestenes and Stiefel, 1952], the LM algorithm [Levenberg, 1944], the Nelder-Mead (NM) method [Nelder and Mead, 1965], and Powell's (P) method [Powell, 1964]. These methods differentiate how the numerical values are updated and whether they incorporate gradient information (BFGS, CG, LM) or not (SA, NM, P). If gradient information is required it has been numerically calculated by taking finite differences. The performance of the methods has been evaluated on 15 different test functions using the correct model structure with varying starting values for each of the 50 repetitions. Overall the NM and LM algorithms worked best on the test functions, further strengthening the point of using LM for constants optimization in symbolic regression.

3.3.2 Gradient Calculation

Crucial for the success of constants optimization is an accurate gradient information $\bigtriangledown f$ of the symbolic regression models. In general there are three ways for gradient calculation:

- Numeric differentiation
- Symbolic differentiation
- Automatic differentiation

CO-NLS uses automatic differentiation [Rall, 1981; Griewank and Walther, 2008] for gradient calculation. Numeric differentiation has the disadvantages

of introducing inaccuracies and cancellation effects due to floating point arithmetics and choosing the step width h. Symbolic differentiation on the other hand is not the optimal way for gradient calculations by computer programs and could lead to inefficient execution. These disadvantages are addressed by automatic differentiation that is specifically designed to be performed by computer programs.

Automatic differentiation exploits the chain rule for derivative calculation and either forward or reverse accumulation is applied. Forward accumulation is more straight-forward to implement, because the calculation starts from the most inner function. Reverse accumulation [Linnainmaa, 1976] starts from the outer functions and is thus more difficult to implement, but requires fewer calculations. The same principle is used by backpropagation [Dreyfus, 1962] of errors in multilayer perceptrons, which is a special case of reverse accumulation automatic differentiation.

Before the gradient of symbolic expression trees can be calculated, the trees have to be transformed to a data structure with which automatic differentiation can operate. This can be achieved by one tree iteration, which simultaneously extracts the numeric parameters of the tree that act as starting values for the LM algorithm. An optional extension during this transformation step is that artificial tree nodes for linear scaling are inserted, whose numerical values are optimized by the LM algorithm as well. This achieves a straight-forward hybridization of linear scaling with constants optimization. The reason behind the explicit inclusion of linear scaling is that it is not guaranteed that the symbolic expression tree contains tree nodes that can achieve linear scaling of the output values.

The individual steps of tree transformation for gradient calculation are displayed in Figure 3.3. The same symbolic regression model represented by Equation (3.1) (symbolic expression tree representation in Figure 3.2) is reused. In the first step (Figure 3.3a) artificial linear scaling nodes, highlighted in light blue, are inserted in the symbolic expression tree. The reason for the inclusion of the artificial linear scaling nodes is that the LM algorithm minimizes sum of squared errors (Equation (3.2)) and it is not guaranteed that the model already contains linear scaling parameters. The numeric values were chosen as the neutral elements of addition and multiplication so that the model output is not altered before constants optimization. In the second step (Figure 3.3b) numerical values are extracted into the parameter vector $\beta = [0.65106, -1.3160, 1.5156, -17.619, 1.00, 0.00]$ including the newly introduced values 1.00 and 0.00. Afterwards the gradient according to β (Equation (3.3) is calculated and utilized by the LM algorithm for constants optimization.

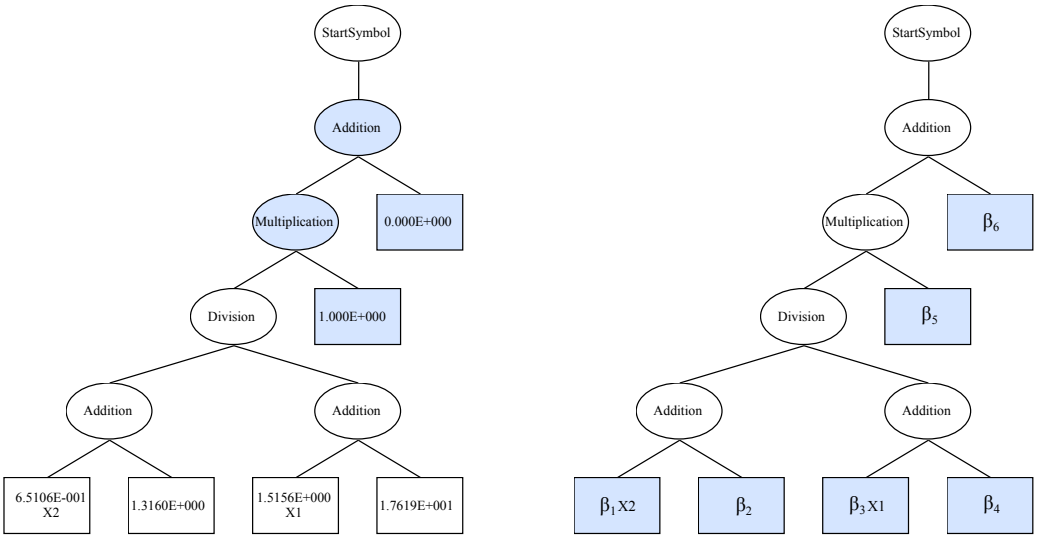


Figure 3.3: Transformation of symbolic expression trees for constants optimization and gradient calculation. In the first step (a) four artificial tree nodes for linear scaling are inserted. Afterwards (b) all numerical values are replaced by the parameters $\beta_1 - \beta_6$.

3.3.3 CO-NLS Algorithm

The two most important building blocks for performing constants optimization by nonlinear least squares (CO-NLS), the gradient calculation by automatic differentiation and the Levenberg-Marquardt algorithm for adapting the numerical values, have been explained in the previous sections. The complete algorithm for constants optimization of a symbolic expression tree is stated as pseudo code in Algorithm 2.

The first few statements account for the tree transformation to make automatic differentiation applicable. There scaling nodes are inserted and numeric values that act as starting values for the LM algorithm are extracted. Afterwards, the LM algorithm is started and for each iteration the gradient at the current values of β is evaluated and the values are updated until the defined stopping criteria is reached. Then the quality of the symbolic expression tree is calculated, according to the defined objective value of the algorithm solving the symbolic regression model. The reason therefore is that the LM algorithm optimizes the mean squared error, but another objective, for example the correlation coefficient R^2 or the mean relative error, could be used to assess the quality of the generated model. Eventually, the optimized numerical values are written back to the symbolic expression tree.

Algorithm 2 Constants optimization of a symbolic expression tree.

```
if Apply linear scaling? then
    Insert artificial scaling tree nodes
end if
Extract numerical values
Transform the tree for gradient calculation
Start the LM algorithm with the extracted values
while Stopping criterion of LM not reached do
    Calculate the gradient by automatic differentiation
    Perform LM iteration
end while
Calculate quality with optimized values
Write optimized values to the according tree nodes
```

The performance of CO-NLS is demonstrated for a linear model (shown in Equation (3.5)) that contains 4 variables ($x_1 - x_4$) and 5 numerical parameters ($w_1 - w_5$). The data, the model has to be adapted to, is generated from the same function without any noise added, assuming the model structure has already been identified correctly. Therefore, data containing 60 observations of the input features x and the target has been created. The correct parameter values for $w_1 - w_4$ have been generated from a uniform distribution $\mathcal{U}[0.0, 10.0]$ and w_5 has been set to exactly zero. The initial starting values for constants optimization have been sampled from another uniform distribution $\mathcal{U}[-50.0, 50.0]$, thus the model has a rather large prediction error.

$$f(x, w) = w_1 \cdot x_1 + w_2 \cdot x_2 + w_2 \cdot x_2 + w_3 \cdot x_3 + w_4 \cdot x_4 + w_5 \tag{3.5}$$

The progression of CO-NLS for each iteration is shown in Table 3.1. In the first row the initial starting values for CO-NLS are stated. The model parameterized with these values has a high mean squared error of approximately 1.40 E+5. The target values that have to be identified are stated in the last row and a correct identification would yield a model without any prediction error.

The values for each parameter w and each iteration show the difference of the current values and the correct values. In detail, *Iteration 0* shows the difference of the initial starting values to the target ones. The following data rows show the progression of CO-NLS after each iteration of the LM algorithm until it has converged. The final identified parameters after five iterations have a difference of at most 1.0 E-14 to the correct values and this model yields a mean squared error of 1.38 E-28.

Table 3.1: Progression of CO-NLS for the linear model displayed in Equation (3.5). The start and target values are displayed in the first and last row respectively. The progress of the algorithm for each parameter w_i is stated for every iteration as the difference between the current and the target values.

	w_1	w_2	w_3	w_4	w_5
Start Values	34.1363	9.2578	−38.3939	−19.0711	0.3228
Iteration 0	27.28	2.15	−40.95	−27.46	0.32
Iteration 1	4.5E−2	3.0E−3	−7.8E−2	−3.9E−2	−2.80
Iteration 2	2.4E−5	9.8E−7	−5.0E−5	−1.7E−5	−1.1E−6
Iteration 3	4.5E−9	−4.5E−11	−1.0E−8	−2.1E−9	−5.5E−10
Iteration 4	2.8E−13	−9.7E−15	−7.8E−13	−4.0E−14	−6.5E−14
Iteration 5	0.0	1.0E−14	0.0	0.0	2.9E−16
Target Values	6.8546	7.1073	2.5583	8.3876	0.0000

Although no noise has been added to the data, the correct values could not be identified to an arbitrary precision due to restrictions in floating point accuracies of the computing system. However, a difference in parameter values below the 10th digit or a prediction error smaller than 10^{-20} can be neglected in practice. The prerequisites to achieve such accurate results are that the model structure is correct, no noise affects the identification of the model parameters, and that the starting values of the methodology are within the range of the global optimal values. In this simplistic example these prerequisites are fulfilled by manually defining the data generating function and the explicit omission of noise. Furthermore, least squares optimization of linear models is a convex optimization problem for which only one global minimum exists and the algorithm converges towards this minimum. For arbitrary symbolic regression models it cannot be guaranteed that the model is linear in its parameters, which is another reason for choosing the LM algorithm for optimizing the numerical values.

3.3.4 Inclusion in Genetic Programming for Symbolic Regression

The described CO-NLS algorithm (depicted Algorithm 2) can be applied to any symbolic regression model that contains only of differentiable functions. Therefore, it is not specific to any algorithm solving symbolic regression problems. In fact, prioritized grammar enumeration (see Section 2.3.2) uses a similar constants optimization method by the LM algorithm for the adaption of the individual model parameters. Another example for the application

of CO-NLS is to adapt the numerical values of symbolic regression models as a post-processing step, after the model has been generated. However, instead of applying CO-NLS as a post-processing step, the direct inclusion in the search for appropriate symbolic regression models generates even more accurate results. This section explains how CO-NLS is integrated in genetic programming for symbolic regression.

The genetic programming algorithm is depicted in Figure 2.1 and stated as pseudo-code in Algorithm 1. If tree-based genetic programming is used for solving symbolic regression problems, every individual encodes a mathematical term encoded as symbolic expression tree, thus making CO-NLS directly applicable. The integration into genetic programming is achieved by applying CO-NLS inside the evaluation step of the algorithm, which assesses the quality of generated models. Before CO-NLS is performed on a specific model, the quality of the model with the initial constants is assessed. This enables genetic programming to discard new numerical values for constants, if the quality of the model would get worse. Although unlikely, this could happen due to misleading gradient information that is for example caused by asymptotes in the model's response. Another possible cause would be differing optimization objectives of CO-NLS and the general symbolic regression problem. Due to the use of the LM algorithm in CO-NLS it minimizes the mean squared error, but the objective of the symbolic regression problem could be to generate a model that is optimal with respect to the mean relative or absolute error.

Additionally to calculating the quality twice (once before and after constants adaption) CO-NLS performs several gradient and model evaluations inside the LM algorithm. This leads to an overhead when compared to genetic programming solving symbolic regression problems without constants optimization. Therefore, it is possible to specify whether all available data points or only a specific subset should be used by CO-NLS to reduce the computational effort. This behavior is specified by the *rows* parameter that determines the relative amount data points that are used by CO-NLS and is set to 100% per default. If the parameter value is smaller, a different subset of the data points is sampled before each execution of CO-NLS.

Another parameter that heavily influences the computational effort is the probability of CO-NLS to be applied to a model. The default value is again 100%, indicating that CO-NLS is performed for every model in the genetic programming population. If CO-NLS is applied probabilistically, the decision if a model is subject to constants optimization is decided for each model individually.

Improvements

The effects of using CO-NLS in genetic programming solving symbolic regression problems are highlighted by an exemplary algorithm execution solving the noise-free *Poly-10* problem (see Section 3.4.2 for further details). The genetic programming algorithm uses a population size of 500 and is stopped after 100 iterations. The quality of a symbolic regression model is assessed as the squared correlation coefficient R^2, thus linear scaling is enabled in CO-NLS and it is applied to every individual and uses every present data point.

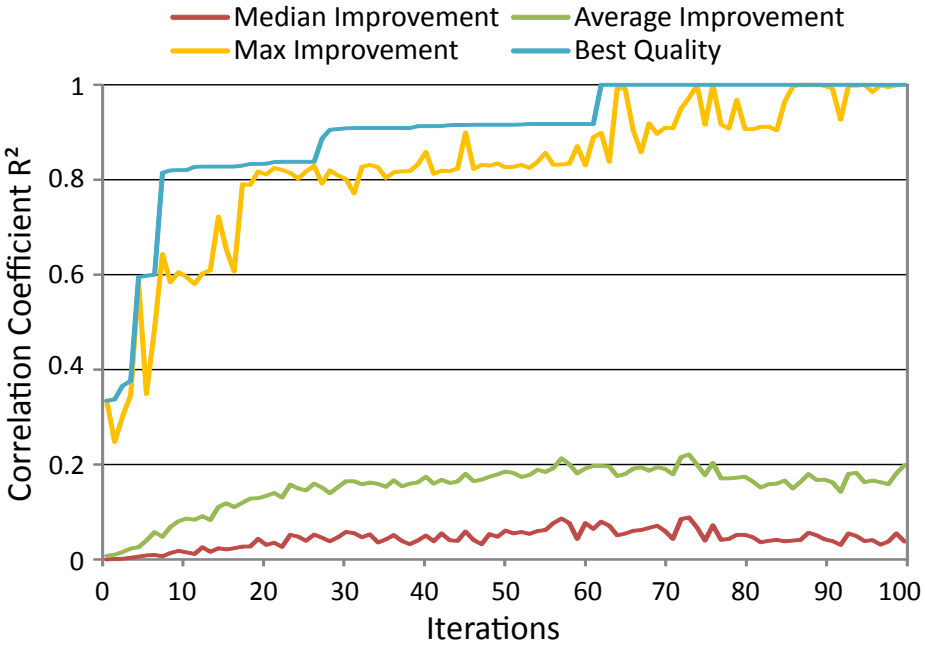


Figure 3.4: Improvements obtained by CO-NLS integrated in genetic programming for an exemplary algorithm execution.

The progression of the objective values in terms of the squared correlation coefficient R^2 and aggregates of the improvements obtained by CO-NLS are displayed in Figure 3.4. This exemplary algorithm execution solves the presented problem and generates a model with perfect predictions, indicated by reaching a value of 1.0 for the best quality. More interesting are the improvements due to constants optimization. The objective value of every generated model in each iteration is assessed before and after the application of CO-NLS and the difference is plotted in the chart. The average and median improvement fluctuate below 0.2 and 0.1 respectively.

However, the best improvement is only slightly below the current best quality. This shows that just by adaption of the numerical values of a model its prediction accuracy can be drastically increased. Furthermore, the assumption and the reasoning to use constants optimization, namely that high quality models are not identified by the algorithm correctly due to misleading numerical values, is at least verified for this exemplary algorithm execution.

Unfortunately, it is not possible to compare the effects of CO-NLS on such detailed level for each generated model to what would happen without constants optimization. The reason therefore are the stochastic aspects of genetic programming, so that every slight change in terms of individuals or their fitness value changes the generated models completely. If only one fitness value is different other parents are selected, resulting in other created child individuals and the whole population in further iterations changes. Therefore, the improvements of CO-NLS can only be studied by performing multiple repetitions of the algorithm with and without constants optimization and comparing the obtained results. Such results are presented in Section 3.4.

Implementation Details

The methodology for constants optimization has been implemented in HeuristicLab [Wagner et al., 2014] described in Section 2.2 and is available since HeuristicLab version 3.3.8. CO-NLS has been implemented directly as an evaluator for symbolic regression, the `ConstantOptimizationEvaluator`[1]. This evaluator handles the parameterization of the LM algorithm, the transformation of the symbolic expression trees for automatic differentiation, the inclusion of linear scaling, and the calculation of the objective values. In addition it provides a static application programming interface (API) so that the CO-NLS could be reused by other parts of the framework.

CO-NLS is reused by nonlinear regression in HeuristicLab, which parses an arbitrary model structure in text form and adapts all numeric constants in the model to the data at hand. A prerequisite is again, due to the use of LM and automatic differentiation, that all functions of the model are differentiable. Another example that uses CO-NLS is the simplification view that enables post-processing of symbolic regression models. The simplification view, shown in Figure 3.5, allows mathematical transformation of the models, assessment of the evaluation impact of model parts, and optimization of constants while directly observing the results [Affenzeller et al., 2014].

[1] https://src.heuristiclab.com/svn/core/trunk/HeuristicLab.Problems.DataAnalysis.Symbolic.Regression/3.4/SingleObjective/Evaluators/SymbolicRegressionConstantOptimizationEvaluator.csf [Accessed 10-Jan-2018]

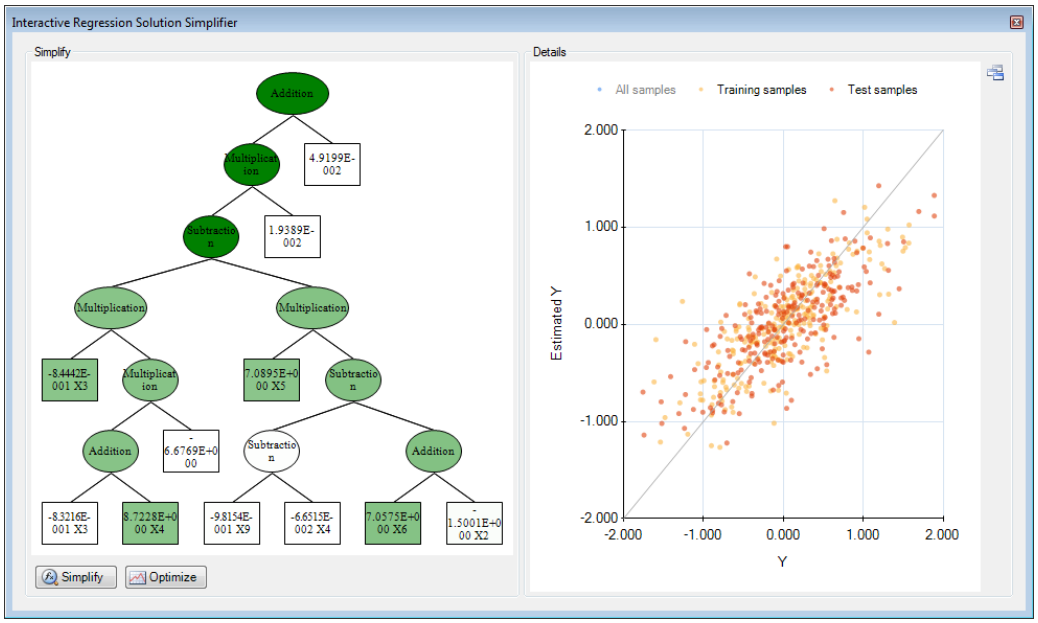


Figure 3.5: Simplification view for symbolic models in HeuristicLab.

The LM algorithm has not been implemented anew, instead the C#version of the free edition of ALGLIB[2] has been included. ALGLIB is a high quality numerical analysis and data processing library and provides a robust implementation of the LM algorithm, which is used for constants optimization.

Automatic differentiation is provided by the AutoDiff library[3]. AutoDiff has been implemented for research on geosemantic snapping [Shtof et al., 2013] and is provided as open-source library for C#. It provides fast and accurate gradient calculation in linear time by reverse accumulation automatic differentiation and is easily extensible for custom functions.

2 http://www.alglib.net [Accessed 21-Dec-2017]
3 https://github.com/alexshtf/autodiff [Accessed 27-Dec-2017]

3.4 Experiments

After the description of constants optimization and exemplary, preliminary results to underline the concepts, this section contains experiments to investigate the effects of CO-NLS when solving symbolic regression problems with genetic programming.

3.4.1 Comparison with Linear Scaling

Keijzer [2003] presented linear scaling and compared the results with the constants optimization approach by Topchy and Punch [2001] and observed the following:

> ... the magnitude of the differences indicate that scaled GP performs at the very least as good as the use of gradient descent. It is however left as future work to compare simple linear scaling as is done here with more involved coefficient fitting methods ... [Keijzer, 2003]

As highlighted in the quote, linear scaling performs as good or better than the gradient descent approach. Although the cited publication does not compare the gradient descent methodology directly to linear scaling, both methods were compared to standard genetic programming and their particular improvements have been collected. We pick up what has been left as future work and compare CO-NLS with standard GP and linear scaling.

In the original paper the performances of the different algorithm variants have been evaluated on five exactly defined benchmark problems that are listed in Table 3.2. These problems are numbered the same way as they are in the original publication. Each problem contains two features (x, y) and does not incorporate any noise. 20 samples are allowed to be used for training that are generated from a uniform distribution $\mathcal{U}[-3, 3]$. The 3712 test samples have been sampled by $E[-3, 0.1, 3]$ that generates all numbers between $[-3, 3]$ with a step size of 0.1. The only difference to the original definition is that we increased the step size from 0.01 to 0.1, which reduces the size of the test partition by a factor of 100 (originally the test partition contained $371,200$ data points).

Table 3.2: Definition of Keijzer benchmark problems.

Problem	Function	Training	Test
Keijzer-11	$f(x, y) = xy + \sin((x-1)(y-1))$	20 samples	3721 samples
Keijzer-12	$f(x, y) = x^4 - x^3 + y^2/2 - y$	20 samples	3721 samples
Keijzer-13	$f(x, y) = 6 * sin(x) * cos(y)$	20 samples	3721 samples
Keijzer-14	$f(x, y) = 8/(2 + x^2 + y^2)$	20 samples	3721 samples
Keijzer-15	$f(x, y) = x^3/5 + y^3/2 - y - x$	20 samples	3721 samples

The results reported by previous publications and achieved by gradient descent and linear scaling are listed as the mean squared error on the training partition in Table 3.3. The first two result columns show the average of 10 repetitions obtained by genetic programming (unscaled GP) and genetic programming with gradient descent (HGP), taken from Topchy and Punch [2001]. The two last columns show the average of 50 repetitions obtained by genetic programming (unscaled GP) and genetic programming with linear scaling (scaled GP), taken from Keijzer [2003].

When comparing scaled GP to HGP, it can be observed that the average mean squared error is always lower when performing scaled GP. However, the base line for comparision (unscaled GP) varies significantly between the two genetic programming systems and thus the improvement between unscaled GP and either HGP or scaled GP has to be evaluated. Judging from this limited amount of data it seems that scaled GP performs better on these five benchmark problems. We have reimplemented these problems and based on the already presented results, further investigated the improvements obtained by linear scaling and CO-NLS and compared the results to standard genetic programming.

Table 3.3: Comparison of training performances (mean squared error) on Keijzer benchmark problems. Unscaled GP states the base lines against gradient descent (HGP) [Topchy and Punch, 2001] and linear scaling (scaled GP) [Keijzer, 2003] are compared to.

Problem	unscaled GP	HGP	unscaled GP	scaled GP
Keijzer-11	0.80	0.47	0.37	0.11
Keijzer-12	2.18	1.03	2.80	0.10
Keijzer-13	6.59	5.98	2.74	0.43
Keijzer-14	4.41	4.06	0.47	0.001
Keijzer-15	0.78	0.27	1.60	0.12

We could not perform exactly the same experiments as reported in these two discussed publications, but we tried to replicate the experiments as closely as possible. The reasons therefore are that subtle changes in the genetic programming implementations could lead to significant changes in the obtained results. For example, we use a weighting factor for each variable (see Section 3.1.2), handle divisions by zero differently, or apply other mutation operators to the symbolic regression solutions and all of these differences change the genetic programming system. Furthermore, Keijzer [2003] performed the experiments using a steady-state algorithm with specialized handling of constants.

We allowed the genetic programming system to build symbolic expression trees with a maximum tree length of 100 and a maximum depth of 50. PTC2 [Luke, 2000a] has been used for tree creation with a function set $\mathcal{F} = \{+, -, *, /, \sqrt{}\}$, where all functions are binary functions except the unary square root function. The terminal set is $\mathcal{T} = \{w * v, r\}$, where v is the variable symbol (for these benchmark problems either x or y), w the weighting factor for variables, and r the ERC symbol representing constant values.

The genetic programming algorithm uses rather standard parameter settings. The population size has been set to 500 and 50 generations have been performed resulting in $25,000$ solution evaluations. Tournament selection with a group size of 5 has been used for parent selection and a 100% crossover rate with standard subtree crossover for child creation. The mutation rate has been 25%, where either tree shaking (adaption of constants or weights), replace or remove branch, or change node type mutation has been performed. Additionally, the best individual (elite) has been passed to the next generation without any modification to achieve a steady increase in solution quality.

These described settings have been kept the same for all algorithm variants. The difference among the variants is the solution evaluation, where either the mean squared error (MSE unscaled), the mean squared error with linear scaling (MSE scaled), the coefficient of determination with linear scaling (R^2 scaled), or the coefficient of determination with optimization of constants including linear scaling (CO-NLS) has been used. CO-NLS performs additional evaluations of the solutions to adapt the numeric constants and therefore has an increased computational effort. Therefore, we also included one additional configuration (CO-NLS fast) that performs constants optimization, but works with a reduced population size of 100 to account for this additional effort.

Table 3.4: Training performance in terms of the mean squared error as the average and standard deviation (±) of 50 algorithm repetitions.

Problem	MSE unscaled	MSE scaled	R^2 scaled	CO-NLS scaled	CO-NLS fast
Keijzer-11	0.57 ± 0.75	0.10 ± 0.03	0.17 ± 0.04	0.00 ± 0.00	0.01 ± 0.01
Keijzer-12	30.96 ± 57.50	1.42 ± 0.95	1.33 ± 1.26	0.00 ± 0.00	0.12 ± 0.24
Keijzer-13	4.21 ± 1.34	2.28 ± 0.92	2.61 ± 0.91	0.00 ± 0.00	0.18 ± 0.44
Keijzer-14	0.15 ± 0.06	0.05 ± 0.03	0.18 ± 0.09	0.00 ± 0.02	0.03 ± 0.05
Keijzer-15	1.70 ± 1.48	0.21 ± 0.07	0.22 ± 0.06	0.00 ± 0.00	0.05 ± 0.07

For each algorithm variant 50 repetitions have been performed and the best solution on the training partition is returned as result. The results are afterwards aggregated and the averages and standard deviations are reported. Table 3.4 reports the performance of the best solutions on the training partition as the mean squared error between the model estimates and the target values. The results obtained by standard genetic programming without any improvements (MSE unscaled) are in a comparable range as the ones obtained by Topchy and Punch [2001] and Keijzer [2003]. The only exception is on the Keijzer-12 problem, where the MSE is much higher due to the large variance of the response values, especially when both present variables are negative the variance increases further.

The improvements obtained by the inclusion of linear scaling (MSE scaled or R^2 scaled) are again very similar to the previously reported values and reduce the MSE by a significant factor. Furthermore, there is almost no difference if the coefficient of determination R^2 is maximized or the mean squared error is minimized, when linear scaling is enabled.

The best results have been achieved by CO-NLS that reduces the MSE to 0.00 for all tested problems. Even the CO-NLS configuration that worked with a fifth of the population size (CO-NLS fast) achieved a smaller MSE and standard deviation than the other algorithm variants.

Table 3.5: Test performance in terms of the mean squared error as the average and standard deviation (±) of 50 algorithm repetitions.

Problem	MSE unscaled	MSE scaled	R^2 scaled	CO-NLS scaled	CO-NLS fast
Keijzer-11	68.99 ± 129.37	67.15 ± 63.00	193.82 ± 221.37	146.13 ± 138.03	100.87 ± 116.63
Keijzer-12	763.62 ± 2271.44	1742.25 ± 2857.81	332.44 ± 1054.95	2907.91 ± 6263.82	2936.63 ± 6869.15
Keijzer-13	126.27 ± 170.73	94.46 ± 108.17	35.69 ± 43.64	247.42 ± 250.10	279.88 ± 270.39
Keijzer-14	3.28 ± 4.33	3.17 ± 4.51	4.74 ± 5.76	2.03 ± 3.83	4.94 ± 5.28
Keijzer-15	98.44 ± 197.69	61.87 ± 141.37	56.32 ± 131.15	102.60 ± 242.98	519.57 ± 596.97

Table 3.5 shows the performance of the best training solution per algorithm execution on the test partition in terms of the averages and standard deviations of the mean squared errors. It is immediately obvious that all of these test results are several magnitudes higher than the results on the training partitions. The difficulty is to identify rather complex mathematical equations from only 20 data points. Although, CO-NLS returns models that estimate the data points on the training partition perfectly, they do not generalize well and are unable to estimate the unknown data points of the test partition accurately.

Furthermore, the standard deviations lies in the same range as the averages of the MSE indicating that the range of the prediction errors is very high. When examining these results more closely, we could verify that these are signs of overfitting and divisions by zero that only occur when evaluating models on the test partition. Therefore, we can conclude that the 20 data points sampled for training the models are too little to be representative for the whole range of the test partition. This might be a reason why Topchy and Punch [2001] and Keijzer [2003] only reported training performances and no test performances.

From these performed experiments we can conclude that linear scaling improves the performance of a genetic programming algorithm regardless of the objective function used. In addition, CO-NLS, even with a reduced population size, outperforms genetic programming with linear scaling. However, all these conclusions only hold true when comparing training performances due to the difficulties when applying the identified models on the test partitions.

3.4.2 Benchmark Problems

Due to the difficulties with the benchmark problems used for the comparison of the new approach with linear scaling, we have chosen a different, more comprehensive benchmark suite to investigate the improvements obtained by CO-NLS. The detailed definitions of the new benchmark problems are given in Table 3.6.

The problems Nguyen-7, Keijzer-6, Valdislavleva-4, and Pagie-1 were suggested as symbolic regression benchmark problems [White et al., 2013]. The Poly-10 problem has been taken from [Poli, 2003], the Friedman-II problem has been introduced in [Friedman, 1991], and the Tower problem has been used in [Vladislavleva et al., 2009].

Table 3.6: Definition of benchmark problems.

Problem	Definition
Nguyen-7	$f(x) = log(x+1) + log(x^2+1)$
Training	20 samples, U[0, 2]
Test	500 samples, U[0, 2]
Keijzer-6	$f(x) = \sum_{i=1}^{x} \frac{1}{i}$
Training	20 samples, E[1, 1, 50]
Test	120 samples, E[1, 1, 120]
Vladislavleva-4	$f(x_1, ..., x_5) = \frac{10}{5+\sum_{i=1}^{5}(x_i-3)^2}$
Training	1024 samples, U[0.05, 6.05]
Test	5000 samples, U[-0.25, 6.35]
Pagie-1	$f(x, y) = \frac{1}{1+x^{-4}} + \frac{1}{1+y^{-4}}$
Training	676 samples, E[-5, 0.4, 5]
Test	1000 samples, U[-5, 5]
Poly-10	$f(x_1, ..., x_{10}) = x_1x_2 + x_3x_4 + x_5x_6 + x_1x_7x_9 + x_3x_6x_{10}$
Training	250 samples, U[-1, 1]
Test	250 samples, U[-1, 1]
Friedman-2	$f(x_1, ..., x_{10}) = 10\sin(\pi x_1x_2) + 20(x_3 - 0.5)^2 + 10x_4 + 5x_5 + \epsilon$
Training	500 samples, U[0, 1]
Test	5000 samples, U[0, 1]
Tower	Real world data
Training	3135 samples
Test	1863 samples

Each of these problems has different characteristics, which make it more or less challenging for an algorithm to create accurate symbolic regression solutions. The problems are approximately sorted by problem difficulty, where the first few problems should be solved without any deviation (if possible) between the model estimations and the data used for training and test.

The Nguyen-7 problem only uses 20 data points for training and yet a mathematical formula containing two terms with logarithms has to be identified. However, a difference to the benchmark problems studied in the previous section is that this function can be rather well approximated by using other mathematical functions.

The Keijzer-6 and Vladislavleva-4 problems define a larger test than training range and thus require the identified solutions to be able to extrapolate correctly. The Pagie-1 problem is difficult to solve accurately, because of the highly nonlinear response. Contrary to the Pagie-1 problem, the Poly-10 problem is built of interactions between two or three variables that are combined linearly without any numerical constants. Nevertheless, it is hard to solve, because of the limited number of available samples. It would become much easier if more samples would be available for training.

The Friedman-2 problem tests the feature selection capabilities of the studied algorithm, because it defines 10 features, but actually there are only five necessary to calculate the response. Additionally, the noise term ϵ is added that is responsible for 4% of the target variance, which further complicates the identification of the correct function. The Tower problem is the only problem containing real world measurements, in this case of a chemical production plant. It contains 26 different measurements as well as an unknown amount of noise and presents similar challenges as the Friedman-2 problem. In general all except the last two problems (Friedman-2 and tower) can theoretically be solved without any prediction errors, because these problems do not contain any noise term.

For all of these problems we limited the available function set to build symbolic regression models from to binary arithmetic functions (+, -, *, /). Therefore, the problems and data generating functions, which contain more complex functions, such as logarithmic or trigonometric functions, have to be approximated.

3.4.3 Genetic Programming Results

In this section the results obtained by standard genetic programming on the defined benchmark problems are presented. The goal is to investigate whether CO-NLS improves the performance of the algorithm in terms of solution accuracy and how well the benchmark problems can be solved by different algorithm variants. The general algorithm settings for genetic programming are listed in Table 3.7.

The restrictions on the generated symbolic solutions are that their symbolic expression tree representation has a maximal tree length of 50 and a maximum of 10 tree levels. These 10 tree levels already account for linear scaling nodes $(a * f(x) + b)$, hence the maximum level for the trees is eight. The function set consists of strictly binary, arithmetic functions and either numerical constants or variables multiplied by a weighting factor are allowed as terminal nodes. The probabilistic tree creator (PTC2) [Luke, 2000a] is used for creating the initial population with an uniform size distribution $U[1, 50]$.

Table 3.7: Standard genetic programming settings.

Parameter	Value
Maximum tree length	50 nodes
Maximum tree depth	10 levels
Function set	binary functions$(+, -, \times, /)$
Terminal set	$constant$, $weight * variable$
Tree initialization	PTC2
Population size	500 individuals
Termination criterion	200 generations
Elites	1 individual
Selection	Tournament selection Group size 4
Crossover probability	100%
Crossover operator	Subtree crossover
Mutation probability	25%
Mutation operator	Change symbol, Single point mutation, Remove branch, Replace branch
Objective function	Maximize R^2
Linear scaling	Enabled
CO-NLS probability	0% | 25% | 50% | 100%
CO-NLS iteration	3 | 5 | 10 Iterations

The settings for the crossover and the mutation operators, as well as their according probability, are the same as in the experiments performed for the comparison of CO-NLS with linear scaling. Again one elite individual is passed to the next generation without any modification and tournament selection is used in the recombination step.

The objective function optimized by genetic programming is the coefficient of determination R^2 that requires linear scaling of the models to be enabled. The reason therefore is that the R^2 is invariant to linear transformations, hence model estimations can be shifted and must be rescaled before they could be interpreted correctly. The population size has been set to 500 and the algorithm performs exactly 200 generations resulting in a total of 100,000 evaluated solutions. The settings are the same for all tested algorithm variants and have been chosen to generate as accurate as possible symbolic regression solutions.

The differing feature between the algorithm variants is whether constants optimization is enabled, with which probability and for how many iterations it is applied. The algorithm variants performing CO-NLS are in the following termed as *CO-NLS probability iterations*, for example *CO-NLS 50% 5 Iterations*. As stated already, constants optimization increases the computational effort for the algorithm, because additional gradient and function evaluations have to be performed to adapt the numerical constants. Nevertheless, the number of evaluated solutions when omitting the numerical parameters, is capped by a maximum of 100,000. However, to ensure a fair comparison additional standard genetic programming variants with an increased population size of 1,000 and 5,000 and adapted tournament group size have been tested. Theses variants termed *GP Pop 1,000* and *GP Pop 5,000* without CO-NLS evaluated a total of 200,000 or 1,000,000 solutions respectively.

Each algorithm variant is executed 50 times for each benchmark problem and the normalized mean squared error (NMSE, also termed fraction of variance unexplained) is reported for the best training solution. The NMSE is calculated as the mean squared error divided by the variance of the target values and can be interpreted as the amount of variance that is not explained by the model. The results are reported as the median and interquartile range (IQR) to be more robust against occasional outliers. For each problem the training performance is listed in the first row (median $\pm$ IQR) and the test performance in the second row. The results are grouped according to the probability of CO-NLS to be applied. The configurations without CO-NLS are listed in Table 3.8, with 25% probability of CO-NLS in Table 3.9, with 50% probability in Table 3.10, and when CO-NLS is applied to all models (100%) in Table 3.11.

Table 3.8: Genetic programming results without CO-NLS.

Problem	GP Pop 500	GP Pop 1,000	GP Pop 5,000
Nguyen-7	0.000 ± 0.000 0.001 ± 0.012	0.000 ± 0.000 0.000 ± 0.009	0.000 ± 0.000 0.000 ± 0.003
Keijzer-6	0.000 ± 0.000 0.020 ± 0.027	0.000 ± 0.000 0.019 ± 0.042	0.000 ± 0.000 0.014 ± 0.020
Vladislavleva-4	0.572 ± 0.131 7.088 ± 19.27	0.461 ± 0.154 9.933 ± 13.65	0.213 ± 0.195 3.287 ± 6.418
Pagie-1	0.096 ± 0.082 0.313 ± 36.25	0.077 ± 0.063 0.238 ± 5.064	0.028 ± 0.047 0.063 ± 0.142
Poly-10	0.162 ± 0.236 0.185 ± 0.240	0.138 ± 0.228 0.180 ± 0.312	0.069 ± 0.005 0.108 ± 0.023
Friedman-2	0.242 ± 0.057 0.238 ± 0.042	0.205 ± 0.095 0.210 ± 0.079	0.128 ± 0.042 0.161 ± 0.990
Tower	0.144 ± 0.025 0.146 ± 0.026	0.135 ± 0.018 0.136 ± 0.023	0.123 ± 0.011 0.124 ± 0.016

Table 3.9: Genetic programming results with 25% CO-NLS.

Problem	CO-NLS 25% Iterations 3	CO-NLS 25% Iterations 5	CO-NLS 25% Iterations 10
Nguyen-7	0.000 ± 0.000 0.000 ± 0.000	0.000 ± 0.000 0.000 ± 0.000	0.000 ± 0.000 0.000 ± 0.000
Keijzer-6	0.000 ± 0.000 0.003 ± 0.003	0.000 ± 0.000 0.001 ± 0.002	0.000 ± 0.000 0.000 ± 0.001
Vladislavleva-4	0.462 ± 0.170 2.327 ± 5.636	0.372 ± 0.170 0.553 ± 3.173	0.309 ± 0.134 0.499 ± 4.926
Pagie-1	0.026 ± 0.057 0.081 ± 0.141	0.020 ± 0.035 0.080 ± 3.664	0.012 ± 0.017 0.111 ± 15.00
Poly-10	0.067 ± 0.081 0.111 ± 0.132	0.065 ± 0.014 0.105 ± 0.039	0.065 ± 0.010 0.112 ± 0.035
Friedman-2	0.100 ± 0.024 0.167 ± 0.999	0.079 ± 0.067 0.137 ± 0.540	0.035 ± 0.034 0.044 ± 0.102
Tower	0.130 ± 0.020 0.135 ± 0.018	0.116 ± 0.027 0.117 ± 0.028	0.106 ± 0.013 0.107 ± 0.020

Table 3.10: Genetic programming results with 50% CO-NLS.

Problem	CO-NLS 50% Iterations 3	CO-NLS 50% Iterations 5	CO-NLS 50% Iterations10
Nguyen-7	0.000 ± 0.000 0.000 ± 0.000	0.000 ± 0.000 0.000 ± 0.000	0.000 ± 0.000 0.000 ± 0.000
Keijzer-6	0.000 ± 0.000 0.001 ± 0.002	0.000 ± 0.000 0.000 ± 0.001	0.000 ± 0.000 0.000 ± 0.000
Vladislavleva-4	0.349 ± 0.145 0.788 ± 5.125	0.284 ± 0.102 0.394 ± 1.182	0.164 ± 0.188 0.250 ± 0.281
Pagie-1	0.017 ± 0.018 0.082 ± 13.58	0.006 ± 0.014 0.107 ± 19.64	0.004 ± 0.014 0.026 ± 18.75
Poly-10	0.057 ± 0.069 0.100 ± 0.115	0.059 ± 0.063 0.102 ± 0.122	0.020 ± 0.063 0.034 ± 0.110
Friedman-2	0.042 ± 0.056 0.066 ± 0.569	0.035 ± 0.001 0.043 ± 0.013	0.034 ± 0.001 0.044 ± 0.018
Tower	0.114 ± 0.023 0.119 ± 0.025	0.105 ± 0.012 0.107 ± 0.016	0.095 ± 0.008 0.095 ± 0.012

Table 3.11: Genetic programming results with 100% CO-NLS.

Problem	CO-NLS 100% Iterations 3	CO-NLS 100% Iterations 5	CO-NLS 100% Iterations 10
Nguyen-7	0.000 ± 0.000 0.000 ± 0.000	0.000 ± 0.000 0.000 ± 0.000	0.000 ± 0.000 0.000 ± 0.000
Keijzer-6	0.000 ± 0.000 0.000 ± 0.001	0.000 ± 0.000 0.000 ± 0.000	0.000 ± 0.000 0.000 ± 0.000
Vladislavleva-4	0.253 ± 0.134 0.331 ± 0.551	0.164 ± 0.154 0.287 ± 0.250	0.061 ± 0.052 0.115 ± 0.075
Pagie-1	0.008 ± 0.017 0.042 ± 9.148	0.005 ± 0.009 0.109 ± 19.95	0.000 ± 0.003 0.000 ± 0.085
Poly-10	0.000 ± 0.066 0.000 ± 0.115	0.000 ± 0.056 0.000 ± 0.115	0.000 ± 0.000 0.000 ± 0.000
Friedman-2	0.035 ± 0.001 0.043 ± 0.017	0.034 ± 0.001 0.043 ± 0.014	0.034 ± 0.001 0.044 ± 0.019
Tower	0.106 ± 0.010 0.107 ± 0.019	0.093 ± 0.010 0.096 ± 0.014	0.088 ± 0.008 0.087 ± 0.009

The Nguyen-7 problem is solved almost perfectly by all algorithm configurations. Although the algorithms cannot build solutions that model the data generating function exactly, because logarithms are not included in the function set, highly accurate estimates for the range of the input feature x are built and no significant difference between the algorithm variants is detected.

The algorithms applied to the Keijzer-6 problem show a similar performance as the Nguyen-7 problem and a NMSE of 0.0 is reached by all algorithms on the training partition. The Keijzer-6 problem evaluates the extrapolation capabilities of the models and all three algorithms without constants optimization show slight deviations on the test partitions. However, if constants optimization is enabled, regardless of how often and for how many iterations, the problem is solved almost always.

The first more revealing results are obtained on the Vladislavleva-4 problem. Comparing the training performances highlights that algorithms with a larger population size (GP Pop 1,000 and GP Pop 5,000) achieve significantly better results. A similar picture is revealed when comparing the algorithm variants with CO-NLS, where the more iterations are performed and the higher the probability of CO-NLS the better results are achieved. The algorithms without CO-NLS produce highly overfit solutions, which is indicated by median NMSEs values larger than 1.0 and the wide interquartile range (IQR). This holds also true for CO-NLS 25% and CO-NLS 50%, but the phenomenon is not that excessive, which is indicated by much smaller median NMSEs. However, the IRQ is still rather large, revealing that overfitting is present as well. Only when CO-NLS is applied to all models the results on the test partition get better.

Almost all algorithms on the Pagie-1 problem show signs of overfitting by large IQRs on the test partition. The reason therefore is that a highly nonlinear fraction has to be identified. Again the training and test results get better the more constants optimization is applied. The only configuration that solves this problem reasonable well is CO-NLS 100% 10 Iterations.

An interesting problem for the investigation of the effects of constants optimization is the Poly-10 problem, because it is hard to solve and it does not contain any numerical values. The question is if it is beneficial to apply CO-NLS although no numerical values have to be identified. Interestingly the results obtained by genetic programming without constants optimization are rather poor with test NMSEs of 0.1 or higher. While only slight improvements are observed for CO-NLS 25%, this changes when more constants optimization is applied. Especially, CO-NLS 50% and 100% for 10 iterations almost always solve the Poly-10 problem perfectly.

The last two problems, Friedman-2 and Tower, are the only problems that cannot be solved exactly, because noise is included in the data. While the

noise level is rather small for the Friedman-2 problem, it is unknown for the Tower problem and both problems contain variables unrelated to the output.

When the results obtained on the Friedman-2 problem are analyzed, it becomes obvious that the results improve significantly when CO-NLS is enabled. While the best median NMSE of the algorithms without CO-NLS (GP Pop 5,000) is in the range of 0.16, the algorithm applying the least amount of constants optimization (CO-NLS 25% 3 Iterations) has a similar performance, although the population size is only a tenth. When the amount of applied constants optimization is increased, the error of the identified models on the training and test partition is reduced until the error reaches the noise level.

The results on the Tower problem look at a first glance all very similar without any larger difference across the algorithm configurations. However, the Tower problem contains real-world data including noise and every percent decrease in the NMSE relates to a decrease in the relative error by one percent, which can be significant when a model is applied in a production system. Furthermore, multivariate linear regression (method of least squares) achieves a training NMSE of 0.117 and a test NMSE of 0.110. All results with larger errors can be considered as low quality results, because a standard linear model that can be easily trained, performs better and the advantage of symbolic regression that it can include nonlinearities is not exploited. This is the case for all configurations without CO-NLS and also for the ones with little (CO-NLS 25% or 50%) probability and little iterations (three or five). Only the configurations with a higher likelihood of CO-NLS to be applied for more iterations are able to outperform multi-variate linear regression. This shows that nonlinear parts of the model are beneficial and reduce the prediction errors of the model, but are hard to identify correctly.

A last analysis that we performed is the success rate for problems that do not incorporate any noise. We defined a success as when a model for a benchmark problem is created, whose prediction error in terms of the NMSE on the test partition is smaller than 0.001. This can be interpreted as the model explains more than 99.9% of the variance of the target variable, because the NMSE is scaled by this variance. The success rate then defines in percent how many times a success for the problem at hand can be achieved.

Figure 3.6 shows the success rate for each algorithm variant on the noise-free benchmark problems. The algorithm variants are grouped according to the probability of CO-NLS to be applied; red - no CO-NLS, yellow - CO-NLS 25%, green - CO-NLS 50%, and blue CO-NLS 100%. For each group the intensity represents the effort while solving the problem, the varying population sizes if CO-NLS is not applied or the number of iterations during the optimization of constants.

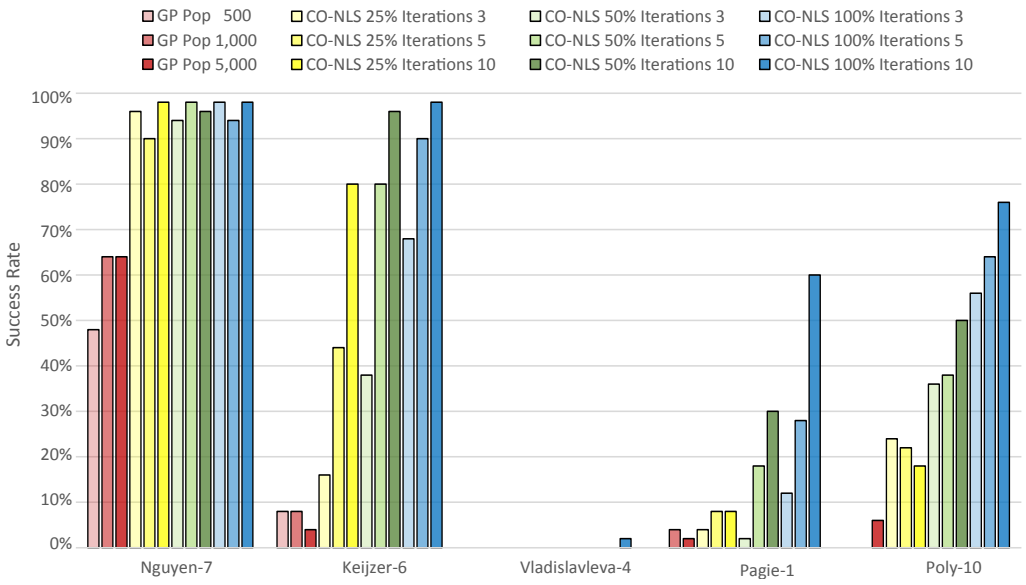


Figure 3.6: Success rates of genetic programming for noise-free benchmark problems with varying population size, CO-NLS probabilities and iterations.

Overall, it can be concluded that CO-NLS outperforms standard genetic programming without constants optimization if such a strict success criteria is used. As seen in the detailed results, the easiest problem to achieve a success on is the Nguyen-7, which is almost always solved if CO-NLS is applied and about 60% of the time otherwise.

The same can be observed for the Keijzer-6 problem, where the success rates for standard genetic programming is on a lower level. A similar picture is depicted for the Pagie-1 and Poly-10 problem, but with reduced success rates for the CO-NLS configurations as well. Interestingly, regardless of the usage of CO-NLS, the Vladislavleva-4 problem can only be solved once, which can be interpreted as fluke. Therefore, it can be concluded that the algorithm configuration that has been used is not able to solve this problem sufficiently.

In conclusion the following observations can be made for each problem of the tested genetic programming variants:

- CO-NLS achieves higher success rates than algorithms without it
- The more iterations of CO-NLS are performed the better the results
- A higher the probability of CO-NLS yields a higher success rate

3.4.4 Offspring Selection Results

In the last section, results for genetic programming with and without constants optimization are presented. All tested algorithm variants have been configured to evaluate exactly 100,000 different models, because 500 models are in the population that is evolved for 200 generations. A drawback of standard genetic programming is that it stops exactly after those 200 generations are calculated, not considering if an improvement would still be possible.

The problem of stopping the algorithm when a number of iterations has been performed is mitigated when offspring selection [Affenzeller and Wagner, 2005] is added to genetic programming. Offspring selection (OS) has been originally developed for genetic algorithms, but due to the close relationship of genetic programming and genetic algorithms it can be easily transfered to genetic programming.

Offspring selection (OS) adds an additional selection step after child creation to genetic programming. After the creation of a new child individual its quality is evaluated and compared to that of its parents. Only if the quality of the child surpasses the quality of the parents, weighted by a comparison factor, it is included in the next population. Otherwise the newly created child is discarded and a new one is generated that is again evaluated and compared to the parents. This process is repeated until the new population is filled. Optionally, a specified number of lucky losers that do not outperform their parents is allowed to participate in building the new population. Strict offspring selection [Affenzeller et al., 2009] that is used for the experiments in this section, is defined as offspring selection with no lucky losers, hence all child individuals must outperform their parents, and that the child quality is compared to the quality of the better parent.

Offspring selection (OS) has the benefits that the selection pressure can be directly calculated. The selection pressure states the ratio between the number of created child individuals (including discarded children) and the population size. Therefore, it directly expresses the effort necessary to create a new population. For example, a selection pressure of 8.4 expresses that for each individual in the new generation on average 8.4 individuals have been created, from which 7.4 have been previously discarded. The selection pressure is commonly used as termination criterion in offspring selection genetic programming. The advantages are that instead of terminating the algorithm after a fixed number of generations, the algorithm stops when no more progress (no better individuals) can be made with reasonable effort.

We performed experiments using the defined benchmark problems with offspring selection genetic programming. This should lead to better and more accurate solutions, because the additional offspring selection step ensures

that the existing genetic material in one generation is beneficially combined in new individuals.

The experiment setup is similar to the one used in the last section. We tested three algorithm variants without constants optimization with varying population sizes (500, 1,000, 5,000) and varying probabilities for CO-NLS (25%, 50%, 100%) with varying number of iterations (3, 5, 10). The parameter settings for offspring selection genetic programming are chosen similarly to the ones used for standard genetic programing and are listed in Table 3.12.

The only differing settings are the termination criterion and the selection operator. Instead of using the number of generations for algorithm termination, the algorithm is stopped when the selection pressure surpasses 100. The other difference is that gender specific selection [Wagner and Affenzeller, 2005] instead of tournament selection is used. The rationale therefore is that the additional selection step introduced by offspring selection reduces the necessity of selecting high quality individuals for reproduction and so a less selective operator can be used for parent selection.

Table 3.12: Offspring selection genetic programming settings.

Parameter	Value
Maximum tree length	50 nodes
Maximum tree depth	10 levels
Function set	binary functions$(+, -, \times, /)$
Terminal set	$constant$, $weight * variable$
Tree initialization	PTC2
Population size	500 individuals
Termination criterion	Selection pressure >= 100
Elites	1 individual
Selection	Gender specific selection
Offspring selection	Strict offspring selection
Crossover probability	100%
Crossover operator	Subtree crossover
Mutation probability	25%
Mutation operator	Change symbol, Single point mutation, Remove branch, Replace branch
Objective function	Maximize R^2
Linear scaling	Enabled
CO-NLS probability	0% | 25% | 50% | 100%
CO-NLS iteration	3 | 5 | 10 Iterations

Table 3.13: Offspring selection genetic programming results.

Problem	OSGP Pop 500	OSGP Pop 1,000	OSGP Pop 5,000
Nguyen-7	0.000 ± 0.000 0.000 ± 0.000	0.000 ± 0.000 0.000 ± 0.000	0.000 ± 0.000 0.000 ± 0.000
Keijzer-6	0.000 ± 0.000 0.004 ± 0.015	0.000 ± 0.000 0.004 ± 0.011	0.000 ± 0.000 0.003 ± 0.032
Vladislavleva-4	0.169 ± 0.139 0.269 ± 0.323	0.099 ± 0.144 0.257 ± 0.262	0.009 ± 0.022 0.017 ± 0.050
Pagie-1	0.048 ± 0.049 0.081 ± 0.617	0.027 ± 0.035 0.060 ± 0.303	0.015 ± 0.015 0.027 ± 0.044
Poly-10	0.163 ± 0.083 0.222 ± 0.135	0.096 ± 0.077 0.131 ± 0.119	0.000 ± 0.075 0.000 ± 0.087
Friedman-2	0.163 ± 0.034 0.177 ± 0.071	0.141 ± 0.038 0.142 ± 0.073	0.087 ± 0.062 0.110 ± 0.084
Tower	0.123 ± 0.008 0.123 ± 0.009	0.120 ± 0.009 0.121 ± 0.010	0.109 ± 0.009 0.109 ± 0.011

Table 3.14: OS genetic programming results with 25% CO-NLS.

Problem	CO-NLS 25% Iterations 3	CO-NLS 25% Iterations 5	CO-NLS 25% Iterations 10
Nguyen-7	0.000 ± 0.000 0.000 ± 0.000	0.000 ± 0.000 0.000 ± 0.000	0.000 ± 0.000 0.000 ± 0.000
Keijzer-6	0.000 ± 0.000 0.001 ± 0.001	0.000 ± 0.000 0.000 ± 0.001	0.000 ± 0.000 0.000 ± 0.000
Vladislavleva-4	0.019 ± 0.061 0.031 ± 0.116	0.008 ± 0.045 0.013 ± 0.086	0.000 ± 0.023 0.000 ± 0.037
Pagie-1	0.012 ± 0.015 0.041 ± 1.807	0.001 ± 0.014 0.012 ± 7.183	0.001 ± 0.008 0.020 ± 15.08
Poly-10	0.018 ± 0.076 0.026 ± 0.140	0.000 ± 0.071 0.000 ± 0.116	0.000 ± 0.020 0.000 ± 0.033
Friedman-2	0.042 ± 0.001 0.043 ± 0.011	0.044 ± 0.063 0.045 ± 0.091	0.041 ± 0.001 0.044 ± 0.128
Tower	0.101 ± 0.016 0.109 ± 0.022	0.092 ± 0.010 0.096 ± 0.015	0.081 ± 0.007 0.083 ± 0.010

Table 3.15: OS genetic programming results with 50% CO-NLS.

Problem	CO-NLS 50% Iterations 3	CO-NLS 50% Iterations 5	CO-NLS 50% Iterations 10
Nguyen-7	0.000 ± 0.000 0.000 ± 0.000	0.000 ± 0.000 0.000 ± 0.000	0.000 ± 0.000 0.000 ± 0.000
Keijzer-6	0.000 ± 0.000 0.001 ± 0.001	0.000 ± 0.000 0.000 ± 0.001	0.000 ± 0.000 0.000 ± 0.000
Vladislavleva-4	0.001 ± 0.030 0.002 ± 0.088	0.000 ± 0.001 0.000 ± 0.001	0.000 ± 0.000 0.000 ± 0.000
Pagie-1	0.006 ± 0.010 0.724 ± 24.49	0.000 ± 0.004 0.008 ± 4.699	0.000 ± 0.000 0.000 ± 0.014
Poly-10	0.000 ± 0.000 0.000 ± 0.000	0.000 ± 0.000 0.000 ± 0.000	0.000 ± 0.000 0.000 ± 0.000
Friedman-2	0.033 ± 0.001 0.042 ± 0.004	0.033 ± 0.001 0.042 ± 0.004	0.033 ± 0.000 0.044 ± 0.028
Tower	0.090 ± 0.013 0.098 ± 0.012	0.083 ± 0.011 0.088 ± 0.011	0.074 ± 0.006 0.078 ± 0.009

Table 3.16: OS genetic programming results with 100% CO-NLS.

Problem	CO-NLS 100% Iterations 3	CO-NLS 100% Iterations 5	CO-NLS 100% Iterations 10
Nguyen-7	0.000 ± 0.000 0.000 ± 0.000	0.000 ± 0.000 0.000 ± 0.000	0.000 ± 0.000 0.000 ± 0.000
Keijzer-6	0.000 ± 0.000 0.001 ± 0.001	0.000 ± 0.000 0.000 ± 0.001	0.000 ± 0.000 0.000 ± 0.000
Vladislavleva-4	0.006 ± 0.016 0.010 ± 0.036	0.002 ± 0.015 0.003 ± 0.021	0.000 ± 0.002 0.000 ± 0.002
Pagie-1	0.001 ± 0.007 0.010 ± 0.309	0.000 ± 0.001 0.000 ± 0.044	0.000 ± 0.000 0.000 ± 0.000
Poly-10	0.000 ± 0.000 0.000 ± 0.000	0.000 ± 0.000 0.000 ± 0.000	0.000 ± 0.000 0.000 ± 0.000
Friedman-2	0.037 ± 0.001 0.051 ± 0.031	0.036 ± 0.001 0.059 ± 0.017	0.036 ± 0.000 0.065 ± 0.127
Tower	0.087 ± 0.012 0.091 ± 0.011	0.079 ± 0.009 0.082 ± 0.012	0.068 ± 0.006 0.071 ± 0.005

The same analysis as in the previous section is performed for offspring selection genetic programming. The experiment settings haven been chosen in a way that the created solutions achieve maximum accuracy. The median and interquartile range of the normalized mean squared error of the best training solution for each benchmark problem grouped by constants optimization probability are listed in tabular form; Table 3.13 includes the results without constants optimization, Table 3.14 with 25% constants optimization probability, Table 3.15 with 50% constants optimization probability, and Table 3.16 with 100% constants optimization probability. The first row for each problem states the training performance (median ± IRQ) and the second row the corresponding test performance.

In general, offspring selection genetic programming produces more accurate results than standard genetic programming. The reasons therefore are twofold. Firstly, due to the additional selection step introduced by offspring selection, the genetic material present in the population is better exploited when building new solutions, because unsuccessful recombinations by crossover and mutation are discarded. Secondly, the algorithm does not stop after a predefined number of generations, but rather when no progress can be achieved anymore.

This is one reason that OS genetic programming without constants optimization is capable of solving the Nguyen-7 and Keijzer-6 most of the times. However, if constants optimization is enabled, the errors are further reduced and almost always zero.

More differences between the algorithm variants can be observed when comparing the achieved results on the Vladislavleva-4 problem, where OSGP Pop 500 and OSGP Pop 1000 achieve a median test error of approximately 0.25 and OSGP Pop 5000 reduces the median test error to 0.017. The results achieved with 25% probability of constants optimization to be applied are in a similar range (regardless of the number of iterations). As early as the probability is increased to 50% or higher, the problem is solved perfectly.

The results of OSGP on the Pagie-1 problem show little errors, but fail to solve the problem and OSGP with smaller population sizes have a larger spread of test errors. The median test error is further reduced by CO-NLS, however if only 25% probability or fewer than 10 iterations are applied, more overfit solutions with large test errors are produced, which is indicated by the large interquartile range. This observations diminish if more and longer constants optimizations is applied. A possible explanation for this phenomena is that due to the high nonlinearity of the problem the gradient information is only helpful when fully exploited.

The Poly-10 problem is solved perfectly by the two variants CO-NLS 100% and CO-NLS 50% although it does not contain any numerical constants

that have to be fitted. The CO-NLS 25% achieves similar results as long as more than 3 iterations of constants optimization are applied. The only configuration with no constants optimization that can compete with that results is OSGP Pop 5000 that utilizes a tenfold larger population size.

The configurations on the Friedman-2 problem without constants optimization achieve a test error between 0.11 and 0.17. The CO-NLS variants reduce this error to 0.04 - 0.06, which is in the range of the noise level of this problem. Interestingly the configurations with 100% probability of CO-NLS to be applied achieve slightly worse results than the others. This is a sign of overfitting that occurs if constants optimization is performed while solving noisy problems.

The results on the Tower real-world problem follow the trend that can be observed on the other benchmark problems, with the exception of the Friedman-2 problem, namely that better results are obtained the more constants optimization in terms of the probability and iterations is applied during the algorithm execution. As already discussed for the results obtained by the genetic programming without offspring selection (Section 3.4.3), a reduction of the median test error on the Tower problem is mostly in the range of percents, but is still significant.

For all noise-free problems we again calculated the success rate as the probability of generating a solution with a test NMSE below 0.001. The results are plotted in Figure 3.7 for all configurations colored according to the probability of CO-NLS to be applied.

Except for the Nguyen-7 problem, the success rates increase significantly when constants optimization is enabled. Even CO-NLS 25% 3 Iterations, the configuration with the least amount of CO-NLS, achieves equal or higher success rates than OSGP-5000, while using a population size of 500. Another observation is that the success rates increase when the number of iterations is increased in the algorithm groups for the same CO-NLS probability. Again there is one exception, CO-NLS 25% 5 Iterations achieves slightly higher success rates than CO-NLS 25% 10 Iterations, but other than that this observation holds.

When comparing the success rates for offspring selection (Figure 3.7) with the success rates of standard genetic programming (Figure 3.6), a drastic increase for all problems can be observed. Every problem is at least solved once with offspring selection, whereas standard genetic programming failed to solve the Vladislavleva-4 problem at all except once. This shows that offspring selection boosts the performance of genetic programming and in combination with CO-NLS is able to produce highly accurate symbolic regression solutions.

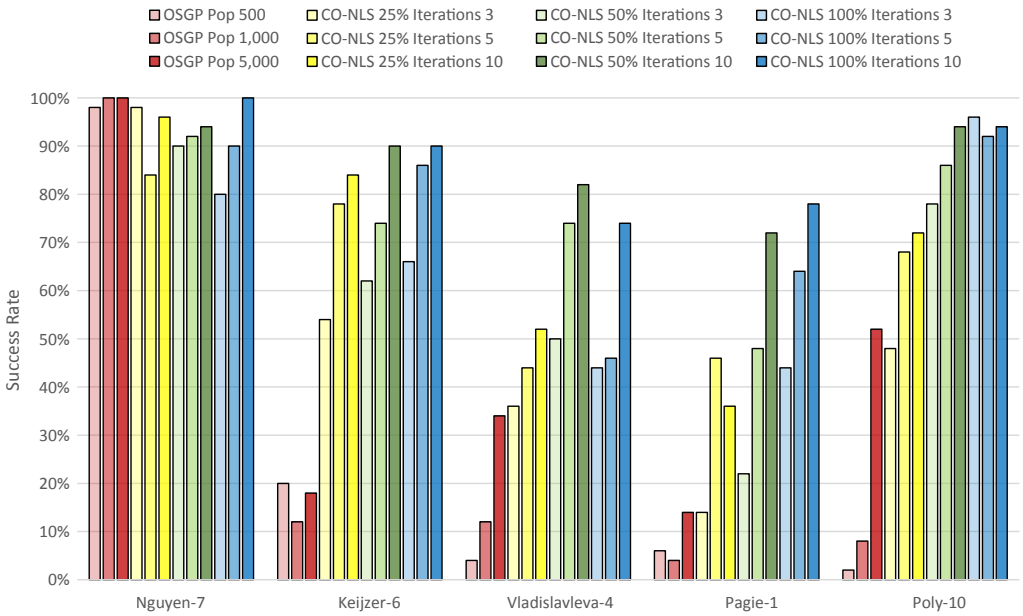


Figure 3.7: Success rates of offspring selection genetic programming for noise-free benchmark problems with varying population size, CO-NLS probabilities and iterations.

In summary it can be concluded that CO-NLS drastically increases the performance of genetic programming when solving symbolic regression problems. This validates our assumption that genetic programming is able to identify the correct model structure and features, but often overlooks such high-quality models due to wrong numerical constants in the model. Even genetic programming with a much larger population size of 5000, which directly translates into more evaluated models, are outperformed by genetic programming with CO-NLS and a population size of 500.

3.5 Concluding Remarks

In this chapter the role of constants in genetic programming solving symbolic regression problems has been discussed and thoroughly evaluated. Constants play a critical role in the generated models, because without appropriate numerical values models cannot produce accurate estimates. In the beginning of genetic programming based symbolic regression constants have only been adapted by random mutation.

In the mean time several ways for the manipulation of constants have been developed. One of the most impactful advancements is linear scaling that removes the necessity of finding the correct range of the model's output by automatically transforming the outputs. More elaborate ways of adapting the constants range from meta heuristics, such as evolution strategies or differential evolution to gradient-based optimization techniques.

We presented a new local optimization approach for tuning the constants of symbolic regression solutions that combines linear scaling, automatic differentiation, and gradient-based minimization of least squares through the Levenberg-Marquard (LM) algorithm. The performance of this new approach has been demonstrated by using a range of benchmark problems and the results are significantly improved when *Constants Optimization by Nonlinear Least Squares* (CO-NLS) is activated.

The main improvement is achieved by using a directed search instead of random mutation for the adaption of numerical values. This results in a paradigm shift for genetic programming based symbolic regression, because genetic programming does not need to find complete models, but rather has to identify promising model structures containing appropriate functions and features and the model is afterwards fitted by CO-NLS to the data at hand.

A disadvantage of CO-NLS is that the LM algorithm converges towards the nearest local optimum depending of the initial starting conditions. This disadvantage is reduced by genetic programming, because it is likely that the same model structure is sampled several times, thus providing different starting conditions for constants optimization. Another helpful technique to escape such local optima is random mutation that is still enabled. However, its role and significance is reduced as the identification of appropriate numerical values is performed by CO-NLS and mutation is only responsible to introduce variations during the search for symbolic regression solutions.

A related issue is that in our implementation every leaf node in the symbolic expression tree representing a model has an associated weighting factor, being this either the constant itself or the weight of a variable. Therefore, the dimensionality of the constants optimization problem is always the number of leaf nodes plus two for the artificial linear scaling terms. For binary ex-

pression trees this results in the number of constants that have to be adapted is half of the total tree length.

Additionally, we do not currently perform any preprocessing of the trees before CO-NLS and thus cannot exclude correlations between the adapted constants or the unnecessary optimization of them. For example, symbolic simplification or pruning can be used to shrink the trees and allow more accurate and faster local optimization of constants.

Another drawback of CO-NLS is the increased computational effort during the search for gradient calculation and optimization. This is especially relevant when the amount of data in the symbolic regression problem increases. An improvement that reduces the computational effort would be to use different gradient-based optimization techniques such as stochastic gradient descent instead of the LM-algorithm.

Nevertheless, the currently employed version of CO-NLS performs reasonably well and improves the accuracy of the symbolic regression solutions generated by genetic programming significantly. We will still evaluate further advancements of this technique and adapt it to our needs to obtain high quality symbolic models.

Chapter 4

Complexity of Symbolic Regression Solutions

In the previous chapter the focus lies on improving the accuracy of symbolic regression solutions. However, the outstanding characteristic of symbolic regression is that accurate solutions are generated as interpretable mathematical formulas. The interpretability is severely hampered if enormous models are created, because too large or complex models are not really interpretable anymore. Therefore, this chapter focuses on the complexity of symbolic regression solutions and how as simple as possible yet accurate models can be created.

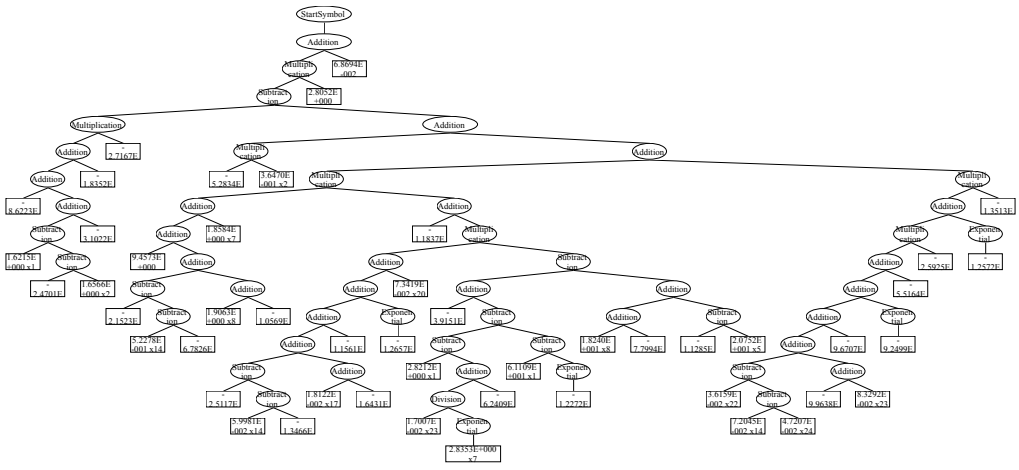


Figure 4.1: Tree representation of a large symbolic regression model consisting of 103 tree nodes.

An example for large symbolic regression models is given in Figure 4.1. The illustrated model predicts the target variable of the Tower problem (defined in Section 3.4.2) and estimates the tower response very accurately. Its symbolic tree representation contains 103 nodes and is the direct result of the genetic programming algorithm. Upon closer inspection it can be observed that several associative symbols, such as *Additions* or *Subtractions* are nested below each other, which can further be simplified by a post processing step.

After post processing, which removes redundant information in the model, the mathematical representation given in Equation (4.1) is retrieved. For a better readability constant numerical values are replaced by c_0 - c_{49}. The model contains 18 different variables x_i and 50 constants c_i and although the mathematical representation is familiar to us the model is still far from interpretable.

$$
\begin{aligned}
f(x) = (&x_1 \cdot (x_1 \cdot (c_0 \cdot x_{23} + c_1 \cdot x_{24} + c_2 \cdot x_8 + c_3 \cdot x_{14} + c_4 \cdot x_{20} + c_5 \cdot x_{22} + \\
&e^{c_6 \cdot x_{10}}) + c_7) \cdot c_8 + e^{c_9 \cdot x_{10}}) \cdot c_{10} + (c_{11} \cdot x_9 + c_{12} \cdot x_{24} + c_{13} \cdot x_8 + \\
&c_{14} \cdot x_{14} + c_{15} \cdot x_7 + c_{16} \cdot x_1 + c_{17}) \cdot ((c_{18} \cdot x_{23} + c_{19} \cdot x_{24} + c_{20} \cdot x_{17} + \\
&c_{21} \cdot x_{14} + c_{22} \cdot x_{20} + c_{23} \cdot x_1 + e^{c_{24} \cdot x_{10}} + c_{25}) \cdot (c_{26} \cdot x_5 + c_{27} \cdot x_{23} + \\
&c_{28} \cdot x_{14} + c_{29} \cdot x_8 + c_{30} \cdot x_{12} + c_{31} \cdot x_6 + c_{32} \cdot x_{22} + c_{33} \cdot x_{19} + \\
&e^{c_{34} \cdot x_4} \cdot c_{35} + \tfrac{c_{36} \cdot x_{23}}{e^{c_{37} \cdot x_7}}) + c_{38}) \cdot c_{39} + x_3 \cdot (c_{40} \cdot x_{24} + c_{41} \cdot x_{14} + \\
&c_{42} \cdot x_1 + c_{43} \cdot x_7 + c_{44} \cdot x_8 + c_{45}) \cdot c_{46} + x_{23} \cdot x_2 \cdot c_{47} + c_{48}) + c_{49}
\end{aligned}
\tag{4.1}
$$

One of the reasons for such large models is that genetic programming increases the size of the individuals in the population during the evolutionary search process. This is often accompanied by an increase in fitness, but also happens without any improvements. The phenomenon of an increase in size without according increase in fitness is termed bloat [Luke, 2000b; Silva and Costa, 2009] and is connected to the presence of introns.

Introns are genome parts that are not expressed and in genetic programming, parts of an individual that do not affect the program output. For example, an intron in genetic programming based symbolic regression is a subtree that is multiplied by zero or another subtree that is divided by a large factor so that its evaluation results in almost zero. In both cases it does not matter how the actual subtree is built and therefore manipulations by crossover and mutation have no or very little effect. The presence of introns is regarded to be beneficial for individuals, because quite often manipulations yield a reduction in fitness. However, if such a manipulation happens inside an intron, this manipulation has no direct consequences with respect to the program output and thus does not affect the individual's fitness. Hence, the accumulation of introns shields an individual from harm-

ful manipulations and the average individual size in a population increases during the evolution.

Another likely reason for the increase in size during evolutionary search is that large individuals in general have better fitness values. Therefore, they are more likely to be selected for reproduction and produce large offspring individuals. As a result fitness-based selection reinforces an increase in size. In symbolic regression bloat and overly large individuals hamper the interpretability of the models and thus diminish one of the major benefits of the methodology.

Bloat is a long studied phenomenon by the genetic programming community and two concerns regarding bloat affect genetic programming. First, the size of individuals in the whole population increase that directly relates to computational effort for the genetic programming system. Second, the result or solution of the algorithm execution is increased and larger than necessary. This is related to the first concern, because if no enlarged individuals are present in the population, the result cannot be enlarged.

Over the years several direct countermeasures to bloat have been proposed [Luke and Panait, 2006]. A first approach in that direction has been undertaken by Koza [1992], who proposed the use of static size limits for the tree depth and length. Without static size limits the genetic programming individuals will grow infinitely and even with limits the individuals will grow until the limit is reached. In general, size limits work by adapting every operator that changes an individual in a way that it adheres to the size limits. For example, in a recombination operation the crossover points are chosen so that the resulting child individual is guaranteed to be within the limits. The performed experiments, which demonstrate the effectiveness of CO-NLS (Section 3.4), also uses static depth and length limits to avoid excessively large models.

Static size limits define the space of possible solutions and directly affect the search of genetic programming. There are much fewer models up to a length of five, compared to all possible models, whose tree representation contains utmost 100 tree nodes. As a result the genetic programming execution is faster the smaller those size limits are sets. These conditions would indicate that size limits should be as strict as possible, but too strict limits result in rather inaccurate models.

The immediate drawback of static limits is that the necessary tree length and depth to produce accurate symbolic regression models cannot be known a-priori and is highly problem dependent. Hence, a common approach is to test several different limit configurations. A better approach would be if the algorithm automatically generates accurate, yet parsimonious models without the necessity of using static size limits.

More advanced methods for controlling the size of the generated models range from dynamic size limits [Silva and Almeida, 2003; Silva and Costa, 2009; Kronberger, 2011], where the size limits are adapted during the algorithm execution depending on the fitness of individuals or the model performance on an internal validation set, or parsimony pressure methods [Luke and Panait, 2006; Poli, 2010] that include the individuals size in the raw fitness values, to controlling the distribution of tree sizes by so-called Tarpeian bloat control [Dignum and Poli, 2008].

All of these methods work reasonably well for controlling the size of individuals in genetic programming and are not specific to symbolic regression. However, the complexity of a symbolic regression model is not solely based on its size, but includes several characteristics, such as the number of occurring features, the different mathematical functions in the model, and how those functions are combined and composed.

In the following section complexity measures for symbolic regression are reviewed and a new one is defined. These complexity measures are later used to perform multi-objective symbolic regression that simultaneously maximizes the model accuracy while model complexity is minimized. As a result the algorithm should automatically adopt the models' complexity to the concrete problems and it becomes unnecessary to specify appropriate size limits.

4.1 Complexity Measures

In the past, several complexity measures for tree-based genetic programming solutions generally and for symbolic regression models specifically have been proposed. The simplest ones applicable to all symbolic expression trees, regardless of what they actually encode, are based on the tree characteristics and are therefore applicable to all kind of genetic programming solutions.

The *Length* defined in Equation (4.2) uses the tree length, the number of all tree nodes, as a measure for complexity. The second complexity measure is the *Visitation Length* and differs from the length by taking the tree shape into account. The visitation length [Keijzer and Foster, 2007], termed *expressional complexity* by Smits and Kotanchek [2005], is calculated by summation over all possible subtrees of the symbolic expression tree (Equation (4.3)). Therefore, shallow tree structures are favored compared to deeply nested ones, because the resulting subtrees contain fewer nodes, which results in a lower complexity value. Another common complexity measure is the number of variables a symbolic regression model contains. The *Variables* complexity measure defined in Equation (4.4) counts the number of features facilitated in the model, but does not take the model size into account.

$$\text{Length}(T) = \sum_{s \in_s T} 1 \tag{4.2}$$

$$\text{Visitation Length}(T) = \sum_{s \in_s T} \text{Length}(s) \tag{4.3}$$

$$\text{Variables}(T) = \sum_{s \in_s T} \begin{cases} 1 & \text{if } sym(s) = \text{variable} \\ 0 & \text{otherwise} \end{cases} \tag{4.4}$$

$s \in_s T$ defines the subtree relation and returns all subtrees s of tree T
$sym(s)$ returns the symbol of the root node of tree s

All these complexity measures do not consider the actual semantics of the model. The *variables* only includes information about the number of features, but not information about the mathematical functions or the model size. There largest advantage is that they can be efficiently evaluated and that they are easily defined and implemented. Their calculation takes a single tree iteration, if caching mechanisms for already calculated subtree lengths are used. Therefore, if any of those complexity measures is integrated in an optimization algorithm, the algorithm runtime is hardly affected.

The drawback of not including any semantic information in the complexity measures is overcome by the *order of nonlinearity* [Vladislavleva et al., 2009]. The order of nonlinearity is based on the mathematical functions occurring in the model as well as the nonlinearity of the model output. It is calculated by recursive iteration of the symbolic expression tree and aggregating the complexity of the individual subtrees. Terminal nodes have a complexity value of either zero or one, depending if the node contains a constant or variable. Internal nodes representing mathematical functions are specifically handled according to an extensive definition. For example, the complexity value of an addition is the maximum of the complexity values of its subtrees, or the complexity of a multiplication is the summation of complexities of its subtrees. A distinguishing feature of the order of nonlinearity is that the definition for calculating the complexity of subtrees includes the minimal degree of a Chebyshev polynomial [Elliott, 1964] approximating the subtrees output, which is an indication of the nonlinearity. As a result the complexity value includes specifics about the functions used and the response of the symbolic regression model. Therefore, it provides an intuitive representation of complexity, but in turn is also more expensive to be evaluated, because for internal tree nodes the approximation of Chebyshev polynomials has to be performed.

Another complexity measure, which takes model semantics into account, is the *functional complexity* [Vanneschi et al., 2010]. The functional complexity is based on the response curvature, which can be interpreted as the deviation of the model response from a straight line. The response curvature for each dimension termed *partial complexity* is approximated as the number of direction changes of the derivative of the model estimates over the input data. Therefore, the model response is sorted according to the dimension the partial complexity should be calculated for and the numerical difference between all neighboring values is stored. The partial complexity is the count of sign changes in that specific difference values and the functional complexity is the average of the partial complexities for all dimensions.

The advantage of the functional complexity is that it reflects the intuitive definition of complexity by approximating the number of infliction points in the model response. Its drawback is that it is highly dependent on the available data points. For example, if a sine function is only evaluated on multiples of π no infliction points are detected and the functional complexity would be zero. Furthermore, it is assumed that all dimensions can be treated independently and the individual results are afterwards aggregated. The functional complexity is not suited and intended to be used within the genetic programming algorithm as optimization objective, but provides an additional useful analysis method for regression models.

4.1.1 Recursive Complexity

Based on the advantages and characteristics of the order of nonlinearity and the functional complexity metrics, we defined a new complexity measure [Kommenda et al., 2015, 2016]. It should be easily calculated, similar to the tree complexity measures, but still include semantics about the symbolic regression models, and should be independent of the actual data points used for training the model. Furthermore, rough estimates of model complexity are sufficient as long as these can be used during multi-objective optimization and steer the optimization algorithm towards simple and parsimonious models. These models should only include complex mathematical functions such as trigonometric, logarithmic, or radical functions if the inclusion gives a significant benefit in terms of prediction accuracy.

$$\text{Complexity}(n) = \begin{cases} 1 & \text{if } sym(n) = \text{constant} \\ 2 & \text{if } sym(n) = \text{variable} \\ \sum_{c \in_c n} \text{Complexity}(c) & \text{if } sym(n) \in (+, -) \\ \prod_{c \in_c n} \text{Complexity}(c) + 1 & \text{if } sym(n) \in (*, /) \\ \text{Complexity}(n_1)^2 & \text{if } sym(n) = \text{square} \\ \text{Complexity}(n_1)^3 & \text{if } sym(n) = \text{squareroot} \\ 2^{\text{Complexity}(n_1)} & \text{if } sym(n) \in (\sin, \cos, \tan) \\ 2^{\text{Complexity}(n_1)} & \text{if } sym(n) \in (\exp, \log) \end{cases} \tag{4.5}$$

$c \in_c n$ defines the child relation and returns all direct child nodes c of node n
indexing is used to refer to the i-th child of a node, i.e. n_1 refers to the first child node of node n
$sym(n)$ returns the symbol of node n

The definition of the *recursive complexity* is given in Equation (4.5). The recursive complexity is calculated by recursive iteration over the symbolic expression tree, where the complexity of each tree node is dependent on the complexities of its child nodes, whose complexity values are aggregated according to specific rules. The complexity of a leaf node is either one or two, depending on whether a constant or variable is encountered. The aggregation rules for complexity values are, if possible, inspired by the mathematical semantics of the symbols.

A major reason for the recursive definition of the complexity metric is that it is heavily dependent on the position of symbols in the expression tree. While the recursive complexity of $\cos(x) = 2^2 = 4$ is rather low, it increases drastically if more complex symbols and functions are the arguments of the cosine function instead of just one variable. As a result the total complexity of a symbolic regression model depends on the level where more complicated functions occur. This is a desired property when performing multi-objective symbolic regression, because then these complex functions are pushed towards the leaf nodes of the tree and thus the interpretability of the symbolic regression model is strengthened.

An alternative definition for the recursive complexity would be to assign the values zero and one to constants and variables respectively. However, a drawback of this definition is that large models that contain many constants would not be penalized and even further multiplications by a constant value would yield a complexity of zero. Therefore, only little parsimony pressure is applied to the models and when this complexity measure is used during optimization, the algorithm would primarily build constant expressions of

varying sizes. Consequently, the constant symbol would gain prevalence in the population and the training of the models would be severely hampered. The same argument explains the presence of the +1 term (Equation (4.5) line 4). One is the neutral element of multiplication and without the +1 term multiplications and divisions containing many constants would yield very small complexity values. This is avoided by defining the complexity of multiplications and division as the product of the by one increased complexities of their child nodes.

The major difference of the recursive complexity to other tree complexity metrics is highlighted by two exemplary symbolic regression models, $f_1(x) = e^{\sin(\sqrt{x})}$ and $f_2(x) = 7x^2+3x+5$. The symbolic expression tree representation of these two models is illustrated in Figure 4.2. When the tree length is used as complexity measure, f_1 is treated as less complex than f_2, because it contains only four tree nodes, whereas f_2 consists of 9 nodes. The same happens if we use the visitation length for complexity calculation; f_1 has a visitation length of ten and f_2 a visitation length of 23, again indicating that f_1 is simpler than f_2.

Figure 4.2 also illustrates the intermediate and final results of the recursive complexity calculation above each node. When the recursive complexity is used as complexity measure, f_2 is treated as simpler than f_1 with complexity values of 17 and 1.15 E+77 respectively. This reflects our intuition that the model f_2 is simpler than the model f_1, whereas according to the tree and visitation length the contrary is true.

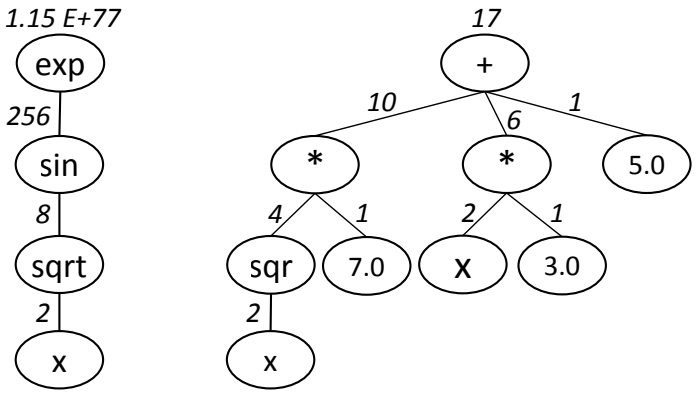


Figure 4.2: Symbolic expression tree representation of $f_1(x) = e^{\sin\sqrt{x}}$ and $f_2(x) = 7x^2+3x+5$, where the calculation steps for the recursive complexity are shown above each tree node.

4.2 Multi-objective Symbolic Regression

Single-objective symbolic regression has already been discussed at length in this thesis and the objective that is optimized is always an error or correlation metric between the model estimates and the true target values. As the name suggests, multi-objective symbolic regression changes the objective from a single one to several ones. In general, one of the objective still measures the errors of the model estimations and at least one additional objective is used, mostly some kind of complexity measure. Therefore, the result of the algorithm solving the multi-objective symbolic regression problem is not a single solution anymore, but a Pareto front of solutions. All solutions of the Pareto front are optimal with respect to the used objectives and the Pareto front represents the tradeoff between different objectives.

An exemplary two-dimensional Pareto front of symbolic regression solutions is illustrated in Figure 4.3 ([Kommenda et al., 2015]). In this figure the models are ordered by the length of the symbolic expression tree encoding the model and for each model the normalized mean squared error (NMSE) of the training and test evaluation is shown. Some models overfit the training data resulting in a test NMSE larger than 1.0 and hence no data point is shown for the test evaluation. This is for example the case for all models with a tree length larger than 40. The test evaluation is shown to indicate that although all models in this figure are Pareto optimal with respect to the tree length and training evaluation, they do not necessarily have good generalization capabilities.

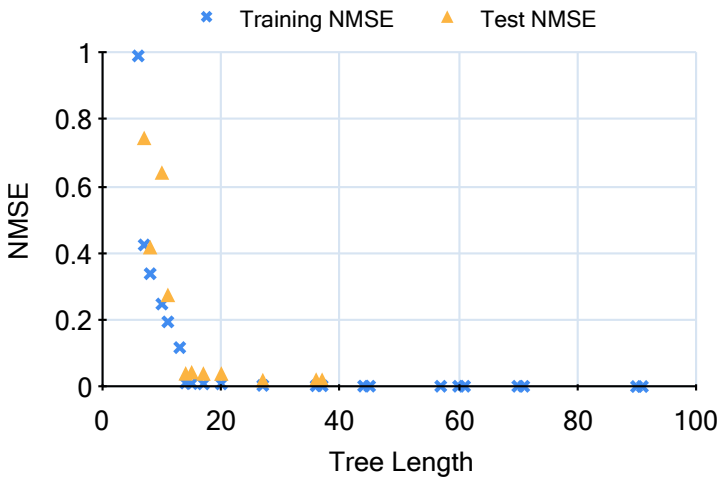


Figure 4.3: Exemplary Pareto-front showing the tradeoff between accuracy in terms of the NMSE and the model length.

Pareto front analysis is a useful tool whenever multiple models have to be analyzed and compared to each other. For example, it is also applicable when single-objective symbolic regression problems are solved by genetic programming by analyzing all models in the final population instead of just considering the model with the smallest estimation errors on the training partition.

However, an obstacle is that due to the single-objective nature of the optimization algorithm only parts of the Pareto front are explored and only models with a high accuracy and hence often more complex models are considered. This effect is highlighted in Figure 4.4, where the minimum, average, and maximum symbolic expression tree length are displayed for each generation of standard genetic programming. While solving the single-objective symbolic regression problem the average tree length increases steadily after an initial decline until it settles slightly below 80. Therefore, if a Pareto front analysis of the final population is performed, mostly accurate yet complex models would be part of the Pareto front. The regions of the Pareto front where very simple and small models with larger estimation errors are located would be almost empty.

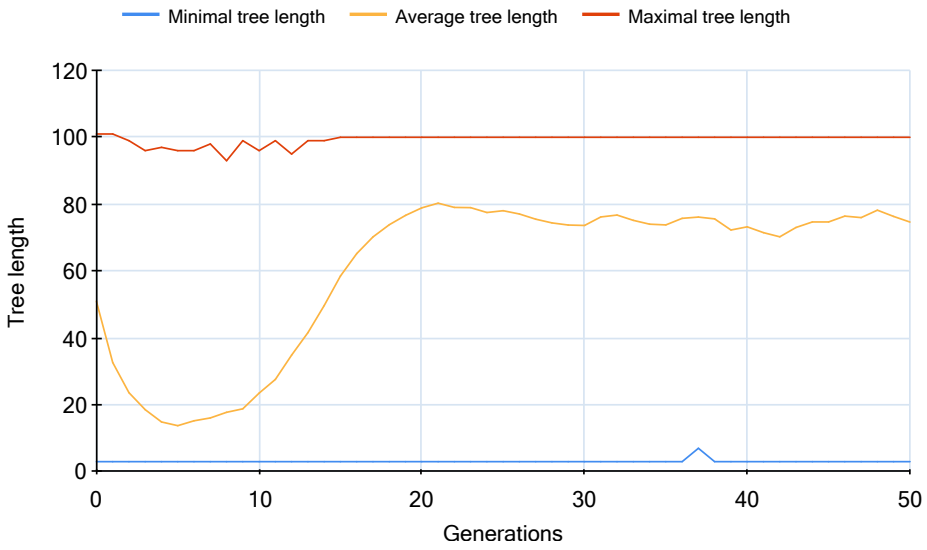


Figure 4.4: Visualization of the minimum, average, and maximum symbolic expression tree length for each generation of standard genetic programming.

Another way of generating more parsimonious regression models while still using single-objective optimization algorithms is to include an regularization term in the objective function. Such a method is for example integrated in FFX [McConaghy, 2011] (see Equation (2.2)), where the L_1 and L_2 norm are used as regularization terms [Zou and Hastie, 2005]. However, a difficulty of such a combined objective function, which includes estimation errors and regularization, is always to balance the weighting factors of individual parts of the objective function. These weighting factors steer the complexity of the generated models and are highly problem dependent.

When performing multi-objective symbolic regression there is no need for weighting factors, because there exist multiple objective functions instead of a single, combined one. Furthermore, all objective functions are independent of each other and treated as equally important. Another benefit is that the whole Pareto front is equally explored during the optimization, which is a key characteristic of multi-objective optimization algorithms.

One of the first designed algorithm for solving multi-objective symbolic regression problems is *ParetoGP* [Smits and Kotanchek, 2005; Kotanchek et al., 2007]. ParetoGP is independent of the concrete objective functions used and maintains an archive of Pareto optimal solutions in addition to the genetic programming population. The search focuses on improving the solutions of the Pareto archive by breeding members of the archive with the most accurate models of the population. After each generation the Pareto archive is updated with the newly generated solution if these are Pareto optimal. ParetoGP commonly uses the Pearson's R^2 correlation as accuracy metric for the solutions and the visitation length of the symbolic expression trees as complexity criterion. However, the order of nonlinearity [Vladislavleva et al., 2010] developed later has also been used in ParetoGP as a objective function measuring the complexity of models.

The goal of this chapter is not to develop a new algorithm for solving multi-objective symbolic regression problems. Therefore, an established multi-objective optimization algorithm that has been designed to solve combinatorial optimization problems is adapted to the specific needs when performing symbolic regression and the most appropriate objective functions for complexity control are evaluated.

Multi-objective optimization for symbolic regression is performed by the nondominated sorting genetic algorithm (NSGA) proposed by Srinivas and Deb [1994]. A drawback of NSGA is that it does not include any kind of elitism, hence a steady increase in the objective values of the solutions in the Pareto front is not guaranteed. Furthermore, an additional parameter for maintaining the diversity of solutions has to be specified and tuned and the runtime complexity of nondominated sorting is rather high.

These disadvantages lead to the development of an improved, more efficient version of the algorithm, the NSGA-II [Deb et al., 2002]. The major improvements of NSGA-II are the ranking of nondominated solutions and crowding distance that guide the algorithms towards an uniformly spread Pareto front. Additionally, elitism is implemented by selecting the individuals for the next generating from the pool of previous solutions unified with the newly created solutions, which is similar to plus selection in evolution strategies [Schwefel, 1981; Beyer and Schwefel, 2002].

Another, further improved version NSGA-III [Deb and Jain, 2014; Jain and Deb, 2014] has been developed since then, which increases the algorithm's capabilities to handle many-objective optimization problems appropriately. Many-objective optimization is characterized by containing more than three objective functions. However, for solving multi-objective symbolic regression problems two objectives, the solution's accuracy and complexity, are sufficient and hence NSGA-II is used for performing multi-objective symbolic regression. Therefore, the published source code[1] of the NSGA-II has been translated into C# and integrated into HeuristicLab [Wagner, 2009; Wagner et al., 2014], as well as the complexity metrics described in the previous section.

4.2.1 NSGA-II Adaptations

We have evaluated the performance of NSGA-II for solving multi-objective symbolic regression problems and the first results were devastating. NSGA-II could not produce accurate solutions and the search for improved solutions has been stuck within a few generations and no further progress has been achieved. The algorithm configuration that has been used is pretty standard; a maximum tree size of 100, a population size of 1,000, termination after 100 generations and the squared Pearson's correlation coefficient R^2 as well as the tree size are used as objective functions. The search behavior is depicted in Figure 4.5, where the minimum, average, and maximum length of the symbolic expression trees representing the models is plotted for one exemplary algorithm execution. Upon further inspection it becomes obvious why no further search progress can be achieved. The average tree length of the population approaches the minimal one rather quickly and after 20 generations the average and minimum tree length are almost equal. Furthermore, the maximal tree length drops quickly and stabilizes at around 10, although the algorithm is allowed to build models up to a size of 100.

[1] http://www.iitk.ac.in/kangal/codes.shtml [Accessed 16-Feb-2018]

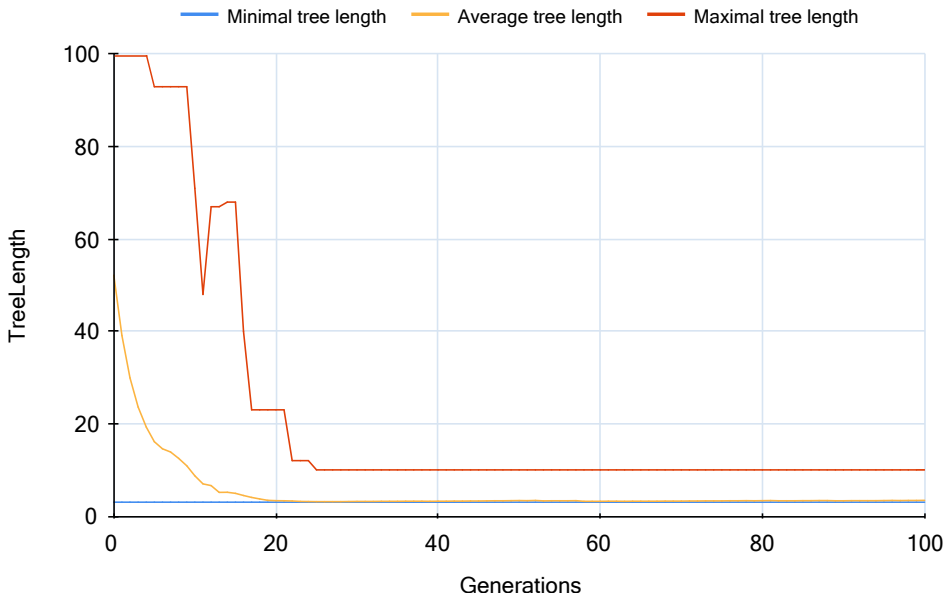


Figure 4.5: Minimum, average, and maximum symbolic expression tree length for each generation of NSGA-II.

This is an indication that the population collapses quickly to few different solutions and as a result no new solutions can be created and the algorithm is stuck. The reason therefore is that in the original version of NSGA-II solutions with exactly equal objective values are treated as nondominated. This is in accordance to the definition of dominance used in multi-objective optimization, which states that a solution x_1 dominates another solution x_2 if and only if the objective values of x_1 for all M objective functions are smaller or equal (assuming minimization) than the according objective values of x_2 and at least one objective value is strictly smaller (Equation (4.6)).

$$x_1 \prec x_2 \iff \begin{cases} \forall i \in 1, ..., M & f_i(x_1) \leq f_i(x_2) \\ \exists j \in 1, ..., M & f_j(x_1) < f_j(x_2) \end{cases} \tag{4.6}$$

This definition of dominance posses a problem when multi-objective symbolic regression problems are solved. The reason therefore is that NSGA-II ranks the solutions of the population into several Pareto fronts. This is done by building the first Pareto front, afterwards the solutions of the first Pareto front are excluded and the remaining solutions are used to build the next Pareto front and so forth until no solutions are left. The rank of the Pareto front a solution is assigned to is called nondomination rank and is the highest influence factor for parent selection in NSGA-II.

The problem when performing multi-objective symbolic regression is now that it is very easy for the algorithm to create a short, but not very accurate solution that is not dominated by any other solution. This solution consists exactly of one variable and nothing else, hence has a tree length of one. If the most appropriate variable is detected, it is not possible to create a smaller or equal sized model that has a better accuracy. However, the same solution can be created multiple times and as these duplicate solutions do not dominate each other. Hence, all of them are assigned to the first Pareto front. Therefore, the first Pareto front is filled during the optimization with very little, duplicate solutions and it is highly unlikely to build larger more accurate solutions and the optimization algorithm is stuck. This behavior is exactly what can be observed in Figure 4.5.

To overcome this issue and still make NSGA-II applicable for multi-objective symbolic regression a relaxed domination criterion is used (Equation (4.7)). Now a solution x_1 dominates another solution x_2 as long as all objective values from x_1 are smaller or equal than those of x_2.

$$x_1 \preceq x_2 \iff \left\{\forall i \in 1, ..., M \quad f_i(x_1) \leq f_i(x_2) \right. \tag{4.7}$$

When using this relaxed domination criterion duplicate solutions dominate each other, because they have exactly equal objective values. As a consequence duplicate solutions are pushed into higher Pareto fronts during the nondominated sorting. However, keeping a single solution of these duplicates is desirable and the first one detected is assigned a domination rank of zero and included in the first Pareto front. Subsequently, all further duplicates are assigned to later Pareto fronts during the nondominated sorting, depending on how many duplicates have been encountered before.

Another possibility to reduce the effect of duplication solutions would be to avoid them at all and only allow unique solutions to be included in the population. Therefore, every child has to be checked whether a copy of it is already included in the population and a new one has to be created it that is the case. However, this alternative creates two additional problems. Firstly, the dynamics of the algorithm are drastically changed, because every generation needs a different amount of individuals to be created until a new population is formed and the available genetic material for recombination is different. Secondly, duplicate detection of solutions with a variable length encoding is not straightforward, because of the complex genotype phenotype mapping. Solutions with the same phenotype need not have the same genotype, thus semantics as well as the representation of a solution have to be compared. This is avoided if relaxed domination is used and the behavior of NGSA-II is more similar to its original form.

Figure 4.6: Minimum, average, and maximum symbolic expression tree length for each generation of NSGA-II with the relaxed domination criterion.

The effect of using the relaxed dominance criterion is shown in Figure 4.6. There the same NSGA-II configuration as in Figure 4.5 is executed and again the minimum, average, and maximum of the symbolic expression tree length at each generation is plotted. Although, a decline in average tree length is present within the first few generations, the population does not collapse into only a handful of different solutions as it was the case with the standard domination criterion and started to rise again towards the end of the algorithm execution. Furthermore, the maximum tree length never drops as low as it was previously the case.

The changes in the symbolic expression tree lengths provide a good indication of what is happening during the algorithm execution. However, to further strengthen the rationale for using a relaxed domination criterion the final population of the two presented algorithm executions are analyzed. In Table 4.1 all solutions of the last population that are members of the first Pareto front and thus Pareto optimal are listed. The first column lists the solutions or their name for larger solutions, where all variables x_1 - x_{20} contain an omitted weighting factor and c represents a constant. The next two columns list the optimization objectives; first the symbolic expression tree length that is minimized and second the correlation (rounded to four decimal places) between the estimates of the solution with the target variable in terms

Table 4.1: Pareto front analysis of NSGA-II with strict and relaxed domination criterion. The solutions, their objective values, and number of occurrences in the Pareto front are shown.

Solution	Objective 1 Length	Objective 2 R^2	Strict dominance	Relaxed dominance
x_6	1	0.4654	704	1
$\log(x_6)$	2	0.4673	217	1
$x_3 \,/\, x_6$	3	0.6193	73	
$x_1 \,/\, x_6$	3	0.6271		1
e^{x_3/x_6}	4	0.6286	5	
$x_{14} \,/\, (x_3 \,/\, x_6)$	5	0.6916		1
$x_3 \,/\, x_6 + x_9 \,/\, x_6$	7	0.7049		1
$x_{20} \,/\, (\log(c) + x_6/x_1)$	8	0.6577	1	
$e^{x_3/x_6} + x_9 \,/\, x_6$	8	0.7062		1
Solution 10	10	0.7160		1
Solution 11	11	0.7176		1
Solution 12	13	0.7311		1
Solution 13	15	0.7324		1
Solution 14	16	0.7344		1
Solution 15	17	0.7360		1
Solution 16	18	0.7417		1
Solution 17	45	0.7494		1
Total			1000	14

of the Pearson's R^2 that is maximized. The third and fourth column show the count of how many times a specific solution is present in the first Pareto front after NSGA-II is finished when using the strict or relaxed domination criterion respectively. A cell is left empty, if that particular solution is not present in the Pareto front. Solutions with an expression tree length of ten or larger are only stated by name because of space restrictions.

Although this analysis consists only of a single algorithm execution for each domination criterion and no parameter tuning has been performed, the overall search behavior is clearly highlighted. The same behavior can be observed for each algorithm execution with slight variations and it underlines the importance of adapting NSGA-II to use a different domination criterion when multi-objective symbolic regression problems are solved.

When using the strict domination criterion the resulting Pareto front contains only five different solutions, but many exact duplicates of those. In total these accumulate to exactly 1000 solutions, which is limited by the

population size of the algorithm and no other solutions are present in final population. In contrast to this, when using the relaxed domination criterion the Pareto front contains 14 different solution with a tree length of up to 45. There are no duplicates in this case, because these are pushed towards later Pareto fronts. When analyzing the whole population there are still duplicates present, especially of small solutions, however there are still 274 different solutions present, which is much more compared to the five different solutions when using the strict dominance comparison. Therefore, from now on all results and analysis are performed using the relaxed domination criterion.

A related aspect that also affects the number of solutions in the final Pareto front is the accuracy of the objective values. Generally, when performing multi-objective symbolic regression one objective describes the complexity and the other objective the accuracy of the solutions. As in the local optimization chapter the squared Pearson's correlation coefficient R^2 is used for evaluating the accuracy. The R^2 lies in the interval $[0, 1]$ and is represented as floating-point number. Due to the floating-point representation of the objective value the Pareto front can get artificially enlarged when solutions with similar quality (up to several decimal places) and varying complexities are present, because these do not dominate each other. Therefore, the relaxed domination criterion has no effect, because the objective values are not equal. However, we are not interested in improvement in terms of solution quality at the tenth decimal place.

An alternative to counteract this phenomenon is to use discretized objective functions that round the objective values to a fixed number of decimal places. This applies a higher selection pressure towards simpler regression solutions as more of them have the same prediction accuracy and are therefore dominated by other solutions because these are either less or equally complex. In addition, the finally obtained Pareto fronts contain in general fewer solutions, because only minor improvements are neglected.

The effects of using a discretized objective function are shown in Figure 4.7, where the number of solutions in the obtained Pareto fronts of 50 NSGA-II repetitions are compared. *Standard* uses the default objective functions, whereas *Discrete* rounds the objective value to three decimal places and size of the Pareto fronts are visualized as box plots. When using the standard objective function that uses the exact objective value the obtained Pareto front has an average size of 22.8 and a median size of 20. In contrast to this, the number of solutions in the Pareto front is halved when using the discretized objective function with an average size of 11.8 and a median size of 10. This gives an advantage because smaller Pareto fronts are easier to analyze, especially if the solutions contained in the front are more diverse.

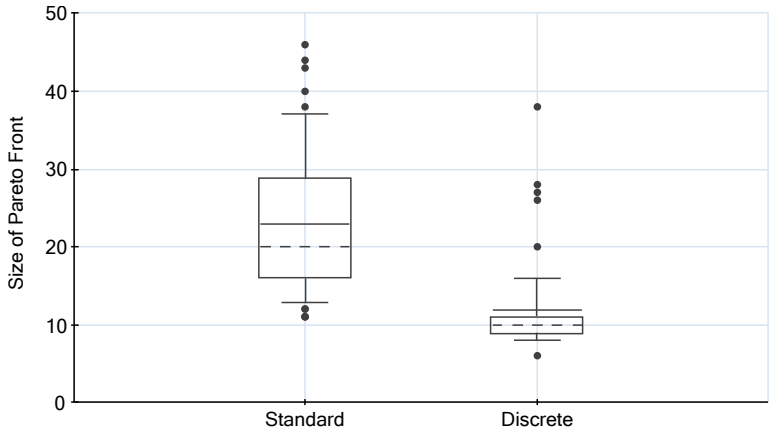


Figure 4.7: Comparison of Pareto fronts obtained by NSGA-II with standard and discretized objective functions.

Furthermore, the two algorithm executions yielding the most accurate regression models are analyzed in detail to illustrate the difference of using exact or rounded objective values. The Pareto fronts of these two algorithm executions are shown in Figure 4.8, where the normalized mean squared error on the training partition and the according symbolic expression tree length of the solutions are displayed. Solutions obtained from NSGA-II with standard objective function are represented by gray circles and solutions obtained using the discretized objective function by black crosses.

The Pareto front obtained using discrete objective values contains only nine solutions, while the other contains 29 solutions. Although the first five solutions with a tree length of up to seven are exactly the same, afterwards the identified Pareto fronts start to differ. While the largest model generated using discrete objective values has a tree length of 37 and a NMSE of exactly $3.02 \cdot 10^{-5}$, the second largest solution has a tree length of only 12 and a NMSE of 0.0023. The difference in accuracy is caused by slightly different numerical values in the shorter solution that, hence these two solutions can be regarded as equivalent.

When using the exact objective values the largest solution consists of 83 tree nodes and yields a NMSE of $5.81 \cdot 10^{-10}$. However, the improvement of an error in the range of 10^{-5} and 10^{-10} can be neglected, because the goal is to generate simple yet accurate regression models. If accuracy maximization at all costs is the goal, it would be better to perform single-objective symbolic regression. Additionally, the algorithm discovered a solution with a tree length of 15 that already yields accurate predictions with a training NMSE of $1.3 \cdot 10^{-4}$, which is the 11th smallest solution.

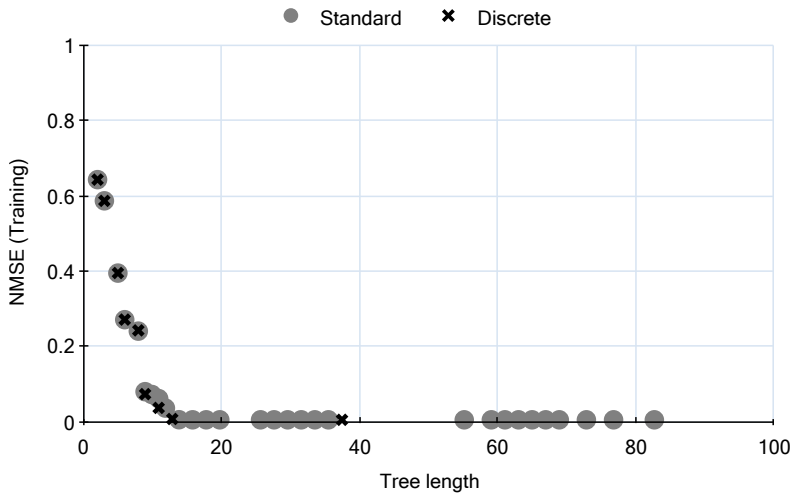


Figure 4.8: Exemplary Pareto fronts obtained by NSGA-II with standard and discretized objective functions.

Upon further analysis it turns out that all solutions from the 11th to the last are equivalent and the increase in the tree length is only used to approximate numerical constants more accurately. This numerical differences in both Pareto fronts could be easily overcome by integrating the local optimization method described in the previous chapter. Overall it can be concluded that the discretized objective functions yield the advantage of generating Pareto fronts with fewer, smaller and simpler solutions without affecting the accuracy of solutions.

4.3 Experiments

After initial tests of NSGA-II for solving multi-objective symbolic regression problems that revealed adaptations, which have to be implemented to change the search behavior of the algorithm, the question remains whether multi-objective symbolic regression is competitive to single-objective symbolic regression in terms of the accuracy of the created solutions. Furthermore, one goal of performing multi-objective symbolic regression is that the algorithm is capable of automatically determining the appropriate length of solutions for the problem at hand. A disadvantage of single-objective symbolic regression based on genetic programming is that the solutions always grow until the specified length and depth limit for symbolic expression trees is reached, which should be avoided when switching to multi-objective symbolic regression.

Another aspect that is of special interest is, how the behavior of NSGA-II changes when different complexity measures are used as additional objective. Furthermore, not only short models in terms of tree length, but simple models with respect to the semantics of the mathematical functions occurring in the symbolic expression trees should be generated. Therefore, a new complexity measure, the recursive complexity, has been defined and is benchmarked against other complexity measures described in the literature. Multi-objective symbolic regression algorithms should also be able to determine the appropriate functions that occur in the models in addition to the appropriate solution length. Therefore, the presence of more complex functions such as trigonometric or power functions in the created solutions is analyzed in the upcoming experiments.

4.3.1 Algorithm Setup

The research questions formulated in the introductory parts of this chapter are answered by performing experiments of different single-objective and multi-objective algorithm configurations that solve symbolic regression problems. We performed three configurations of single-objective symbolic regression with standard genetic programming, where the only different parameter setting that is varied, is the maximum expression tree length. These algorithm configurations are benchmarked against each other and four algorithm configurations of multi-objective symbolic regression solved by the NSGA-II, where the objective function, which measures the complexity of the solutions, is varied.

As both algorithms, standard genetic programming and NSGA-II, are related to genetic algorithms, they have a lot of parameters in common, which have been set to the exact same value. These common and as well as algorithm specific parameter settings are listed in Table 4.2.

The most prominent difference of the algorithm configurations compared to the experiments performed in Section 3.4 is that instead of using only arithmetic functions, trigonometric, exponential and power functions are allowed to be included in the solutions as well. The terminal set consist of either numerical constants or variables that are weighted by a multiplicative factor. The maximum tree depth is set to 50 for any of the algorithms, because unary functions yielding an increased symbolic expression tree depth are enabled and the tree size is restricted by varying maximum lengths.

The initial population is again created by the probabilistic tree creator (PTC2) [Luke, 2000a] and a population of 1,000 individuals is evolved for 500 generations. All algorithms use tournament selection with a group size of five for parent selection before child creation. However, the NSGA-II uses a crowded tournament selection, which does not act on the raw objective value, but on the domination rank and crowding distance. A simple subtree crossover is used for child creation that is applied with 100% probability and afterwards newly created children are mutated with a probability of 25%, where either one of the four described mutation operators is applied to manipulate the solutions. Linear scaling is enabled in all configurations as it enhanced the solutions accuracy. However, further local and constants optimizations are not performed to study the consequences of multi-objective symbolic regression without any side effects.

Standard genetic programming is used for performing single-objective symbolic regression. The three different configurations vary according to the maximum tree length; either 20, 50, or 100 is used to restrict the growth of the symbolic regression trees. The variants are later referred to as *GP Length 20*, *GP Length 50* or, *GP Length 100*. For completeness it is stated that one elite solution is used in the algorithm and the single objective used, is the maximization of the squared Pearson's correlation coefficient R^2. The space of possible solutions is strongly influenced by the maximum tree length. An optimal setting is the smallest length large enough so that a regression solution generating accurate estimates can be built and hence three varying settings are tested.

The adapted nondominated sorting genetic algorithm (NSGA-II) is used for performing symbolic regression, where relaxed domination and discretized objective functions are used. A slight difference to standard genetic programming is that there are no elites in the algorithm, but rather a new generation is built by unifying the existing population with the newly created solutions

and selecting the best solutions out of this union. Hence, the solutions in the Pareto front can be regarded as elites.

The major difference to the single-objective variants is that the maximum tree length is set to 100, whereas it has been varied before. The rationale therefore is that we expect the multi-objective algorithms to automatically identify the appropriate expression tree length and therefore the maximum of the single-objective configuration is chosen as default. In total four different configurations of NSGA-II have been evaluated that differ by the additional objective that judges the complexity of solutions. All variant use a discretized version of Pearson's R^2 as first objective, which measures the solutions' accuracy. The second, complexity related objective is either the length of the symbolic expression tree (*NSGA-II Tree length*, 4.2), the visitation or nested tree length (*NSGA-II Visitation length*, 4.3), the number of variables present in the solution (*NSGA-II Variables*, 4.4), or the newly defined recursive complexity (*NSGA-II Recursive*, 4.5).

Overall seven different algorithm, three single and four multi-objective ones, are evaluated and the resulting consequences of the different variants described. First, artificially defined, noise free benchmark problems and afterwards established, more complicated, noisy problems are used for this purpose.

4.3.2 Results on Benchmark Problems

We defined five artificial benchmark problems that are used to evaluate the different algorithm variants and test the suitability of multi-objective symbolic regression. These problems (*Problem* F_1 - F_5) are listed in Table 4.3, contain no noise and the data is generated by either a five or four dimensional mathematical function. For every problem 500 samples are generated by uniformly sampling each independent input variable from the interval U[-5,5]. Out of these 500 data samples, 100 are used for training and thus learning the solutions and 400 are used as a separate test partition to evaluate the generalization capabilities of solutions.

These benchmark problems have been specifically designed to test whether the algorithms are capable of selecting the appropriate solution length and mathematical functions to model the data. As described in the previous section in addition to arithmetic functions, trigonometric, exponential, logarithmic, as well as square and square root symbols can be integrated in the symbolic expression trees. Each of the defined problems needs arithmetic functions to be present in the created solutions. The problem F_1 additionally contains a squared term and an interacting term between two variables.

Table 4.2: Algorithm settings for single-objective and multi-objective symbolic regression algorithms.

Common parameters	
Maximum tree depth	50 levels
Function set	Binary functions $(+, -, \times, /)$
	Trigonometric functions $(\sin, \cos, \tan)$
	Exponential functions $(\exp, \log)$
	Power functions $(n^2$ and $\sqrt{n})$
Terminal set	*constant*, $weight * variable$
Tree initialization	PTC2
Population size	1000 individuals
Termination criterion	500 generations
Selection	Tournament selection, group size 5
Crossover probability	100%
Crossover operator	Subtree crossover
Mutation probability	25%
Mutation operator	Change symbol, single point mutation,
	Remove branch, replace branch
Linear scaling	Enabled

Single-objective symbolic regression	
Algorithm	Standard genetic programming
Maximum tree length	20 | 50 | 100 nodes
Elites	1 individual
Objective function	Maximize R^2

Multi-objective symbolic regression	
Algorithm	Nondominated sorting genetic algorithm (NSGA-II)
Maximum tree length	100 nodes
Elites	Pareto optimal solutions
Domination criterion	Relaxed domination
Objective functions	Maximize R^2 (discretized) +
	Minimize tree length
	Minimize visitation length
	Minimize variables count
	Minimize recursive complexity

Table 4.3: Definition of noise free benchmark problems

Name	Function	Training	Test
F_1	$F_1(x_1, ..., x_5) = x_1 + x_2 + (x_3 - 2)^2 + x_4 x_5$	100 samples	400 samples
F_2	$F_2(x_1, ..., x_5) = 10 \sin(x_1) + x_2 + x_3 x_4 + x_4 x_5$	100 samples	400 samples
F_3	$F_3(x_1, ..., x_5) = e^{0.7 x_1} + (0.5 x_2 + 2)^2 + {x_3}^2 + x_4 x_5$	100 samples	400 samples
F_4	$F_4(x_1, ..., x_4) = \log((x_1 + x_2)^2)$	100 samples	400 samples
F_5	$F_5(x_1, ..., x_4) = (x_1 + x_2)^2 (x_3 + x_4)$	100 samples	400 samples

Hence, it is either necessary to include to square symbol in the solutions or to create a slightly longer solutions, where the squared term is expanded. F_2 is slightly more complicated, because it contains one trigonometric function and two interacting terms, which are in general harder to identify. F_3 is built out of an exponential, two squared, and one interacting term. Although F_4 has the shortest mathematical representation it's difficulty should not be underestimated, because it contains the logarithm of a squared term, which complicates the correct identification of the formula. The reason therefore is that as long as the squared term is not included, the logarithm of a single variable results in an undefined value due to the negative values of the variables. Furthermore, F_4 uses only two of the four possible variables. The last problem F_5 just consists of a product of terms and if identified correctly the algorithms should not include any trigonometric, exponential, or logarithmic functions at all in the solutions.

The aggregated results of 50 repetitions of each algorithm variant solving each benchmark problem are listed as median ± interquartile range in Table 4.4. A fair comparison between single and multi-objective algorithms is guaranteed, because the most accurate solution on the training partition is picked as the final result and analyzed. The accuracy is evaluated as the normalized mean squared error (NMSE), also termed fraction of variance unexplained, on training and test. The stated length for each solution is the expression tree length after automatic mathematic simplifications and constant folding, which has been applied to get a more parsimonious form of the model. The maximum tree length can be exceeded due to the introduced linear scaling terms in the models that increase the tree length by four (compare Problem F_3 - GP length 20.) For each problem the minimal symbolic expression tree length solving the problem exactly is stated for comparison purposes in bold font. The minimal tree length is a little higher than one would expect, because only binary expression trees can be built and this is also respected when calculating the minimal length.

First the results of single-objective symbolic regression solved by standard genetic programming with the three different maximum tree lengths are discussed. Among these configurations *GP Length 20* performs best in terms of training and test accuracy. A reason therefore is that all benchmark problems can be optimally solved with a maximum solution length of 20. Surprisingly, *GP Length 100* performed worst among all algorithm variants, because one expects that this configuration would include all possible smaller solutions. However, the difficulty for single-objective genetic programming with a fixed evaluation budget is to identify the appropriate length and functions to build accurate solutions, which results in a large variety of solutions indicated by an increased interquartile range of the NMSE on the test partition. This effect, to a lesser extend though, is also visible for *GP Length 50* on problem F_4 or F_5. Thus, it can be concluded that the reduction of the hypothesis space by a smaller maximum tree length helps the algorithm to create more accurate solutions. In summary, *GP Length 20* would be preferred from the single-objective algorithm for solving these benchmark problems, because it creates the smallest solutions and most accurate solutions consistently.

When analyzing the results obtained by multi-objective symbolic regression, the algorithm configuration *NSGA-II Variables* stands out, because it creates the least accurate yet largest solutions. The reason therefore is that the variables complexity measure does not apply any parsimonious pressure during the optimization. The two NSGA-II variants considering the tree or visitation length as objective behave rather similarly, with slight advantages for *NSGA-II Visitation length*, because it applies a higher parsimony pressure. The best results on all problems have been obtained using the recursive complexity measure; only on the problem F_4 the interquartile range of the test NMSE is a little higher compared to the other variants.

The length of the created solutions stays always below the maximum tree length for the single-objective algorithm configurations. As these algorithms have no incentive they create as large solutions as possible as long and the gap between the maximum tree length and solution length results mainly due to the simplifications applied to the final solutions. In contrast, the multi-objective variants (except *NGSA-II variables*) are able to create much smaller solutions than their maximum tree length of 100 would allow. The most difficult problem to detect an appropriate solution length automatically seems to be the problem F_3, where the largest difference between the minimal length of an optimal solution and the obtained median solution length is present. Again the recursive complexity measure yields overall the best results in terms of solution length.

Table 4.4: Training and test accuracy of the best training solution for each algorithm variant in terms of the normalized mean squared error and the solution length. All quantities are expresses as median ± interquartile range. The minimal solution length to solve a problem is stated in bold font.

	Training	**Test**	**Length**
Problem F_1			**12.00**
GP Length 20	0.001 ± 0.028	0.002 ± 0.032	16.0 ± 4
GP Length 50	0.003 ± 0.240	0.003 ± 0.340	34.5 ± 16
GP Length 100	0.023 ± 0.211	0.092 ± 0.535	68.5 ± 31
NSGA-II Recursive	0.000 ± 0.002	0.000 ± 0.004	24.5 ± 40
NSGA-II Tree length	0.035 ± 0.180	0.056 ± 0.392	32.0 ± 59
NSGA-II Visitation length	0.008 ± 0.140	0.013 ± 0.347	34.0 ± 74
NSGA-II Variables	0.089 ± 0.182	0.302 ± 0.505	69.0 ± 20
Problem F_2			**12.00**
GP Length 20	0.000 ± 0.000	0.000 ± 0.000	19.0 ± 4
GP Length 50	0.000 ± 0.009	0.000 ± 0.007	41.0 ± 12
GP Length 100	0.039 ± 0.419	0.053 ± 0.964	86.0 ± 25
NSGA-II Recursive	0.000 ± 0.000	0.000 ± 0.000	19.0 ± 20
NSGA-II Tree length	0.000 ± 0.000	0.000 ± 0.000	29.0 ± 33
NSGA-II Visitation length	0.000 ± 0.000	0.000 ± 0.000	27.0 ± 18
NSGA-II Variables	0.000 ± 0.125	0.000 ± 0.494	56.0 ± 42
Problem F_3			**14.00**
GP Length 20	0.005 ± 0.009	0.008 ± 0.017	23.0 ± 3
GP Length 50	0.002 ± 0.007	0.006 ± 0.016	43.0 ± 10
GP Length 100	0.003 ± 0.107	0.009 ± 0.548	81.0 ± 18
NSGA-II Recursive	0.001 ± 0.009	0.002 ± 0.022	54.5 ± 47
NSGA-II Tree length	0.002 ± 0.010	0.004 ± 0.023	74.0 ± 47
NSGA-II Visitation length	0.001 ± 0.012	0.003 ± 0.025	65.5 ± 49
NSGA-II Variables	0.037 ± 0.144	0.081 ± 0.627	69.0 ± 17
Problem F_4			**5.00**
GP Length 20	0.000 ± 0.000	0.000 ± 0.056	18.0 ± 8
GP Length 50	0.000 ± 0.285	0.102 ± 0.570	41.5 ± 19
GP Length 100	0.125 ± 0.381	0.377 ± 0.944	73.5 ± 23
NSGA-II Recursive	0.000 ± 0.000	0.001 ± 0.082	9.0 ± 3
NSGA-II Tree length	0.000 ± 0.000	0.001 ± 0.002	9.0 ± 0
NSGA-II Visitation length	0.000 ± 0.000	0.001 ± 0.001	9.0 ± 0
NSGA-II Variables	0.000 ± 0.122	0.002 ± 0.445	13.0 ± 56
Problem F_5			**8.00**
GP Length 20	0.025 ± 0.033	0.041 ± 0.045	18.0 ± 4
GP Length 50	0.029 ± 0.032	0.046 ± 0.283	39.0 ± 13
GP Length 100	0.055 ± 0.116	0.846 ± 8.676	81.0 ± 17
NSGA-II Recursive	0.000 ± 0.003	0.000 ± 0.004	24.0 ± 41
NSGA-II Tree length	0.009 ± 0.032	0.038 ± 0.046	27.0 ± 51
NSGA-II Visitation length	0.000 ± 0.032	0.000 ± 0.045	13.0 ± 20
NSGA-II Variables	0.050 ± 0.088	0.287 ± 0.915	67.0 ± 20

Table 4.5: Symbol analysis of the best solution in terms of the affected subtree length expressed as the median ± interquartile range. The minimal affected subtree to solve a problem optimally is stated in bold font.

	Trigonometric Functions	**Exponential Functions**	**Power Functions**
Problem F_1	**0.0**	**0.0**	**4.0**
GP Length 20	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 4.0
GP Length 50	12.0 ± 35.0	2.5 ± 32.0	5.0 ± 22.0
GP Length 100	38.5 ± 88.0	19.5 ± 68.0	25.0 ± 60.0
NSGA-II Recursive	0.0 ± 2.0	0.0 ± 0.0	4.0 ± 12.0
NSGA-II Tree length	5.5 ± 69.0	0.0 ± 22.0	4.0 ± 23.0
NSGA-II Visitation length	0.0 ± 50.0	3.5 ± 17.0	4.0 ± 16.0
NSGA-II Variables	139.5 ± 211.0	47.0 ± 79.0	62.5 ± 82.0
Problem F_2	**2.0**	**0.0**	**0.0**
GP Length 20	4.0 ± 4.0	0.0 ± 0.0	0.0 ± 0.0
GP Length 50	23.5 ± 40.0	0.0 ± 14.0	0.0 ± 3.0
GP Length 100	126.5 ± 196.0	40.5 ± 110.0	32.5 ± 75.0
NSGA-II Recursive	4.0 ± 10.0	0.0 ± 0.0	0.0 ± 0.0
NSGA-II Tree length	6.0 ± 20.0	0.0 ± 0.0	0.0 ± 0.0
NSGA-II Visitation length	6.0 ± 12.0	0.0 ± 0.0	0.0 ± 0.0
NSGA-II Variables	82.5 ± 148.0	12.5 ± 84.0	8.0 ± 44.0
Problem F_3	**0.0**	**2.0**	**6.0**
GP Length 20	0.0 ± 0.0	4.0 ± 4.0	2.0 ± 2.0
GP Length 50	9.0 ± 16.0	4.0 ± 10.0	8.0 ± 9.0
GP Length 100	59.0 ± 104.0	20.0 ± 34.0	28.0 ± 48.0
NSGA-II Recursive	0.0 ± 2.0	6.0 ± 6.0	9.0 ± 16.0
NSGA-II Tree length	0.0 ± 8.0	6.0 ± 10.0	19.0 ± 37.0
NSGA-II Visitation length	0.0 ± 4.0	4.0 ± 8.0	14.0 ± 18.0
NSGA-II Variables	64.0 ± 77.0	27.5 ± 57.0	42.0 ± 81.0
Problem F_4	**0.0**	**5.0**	**4.0**
GP Length 20	0.0 ± 0.0	12.0 ± 8.0	4.0 ± 11.0
GP Length 50	16.0 ± 29.0	23.0 ± 33.0	6.5 ± 28.0
GP Length 100	85.5 ± 162.0	48.0 ± 95.0	25.5 ± 109.0
NSGA-II Recursive	0.0 ± 0.0	5.0 ± 0.0	4.0 ± 0.0
NSGA-II Tree length	0.0 ± 0.0	5.0 ± 0.0	4.0 ± 0.0
NSGA-II Visitation length	0.0 ± 0.0	5.0 ± 0.0	4.0 ± 0.0
NSGA-II Variables	0.0 ± 145.0	9.0 ± 59.0	9.5 ± 84.0
Problem F_5	**0.0**	**0.0**	**4.0**
GP Length 20	0.0 ± 0.0	0.0 ± 4.0	4.5 ± 10.0
GP Length 50	11.5 ± 27.0	0.0 ± 17.0	8.0 ± 17.0
GP Length 100	69.0 ± 154.0	44.5 ± 65.0	30.5 ± 50.0
NSGA-II Recursive	0.0 ± 0.0	0.0 ± 0.0	4.0 ± 4.0
NSGA-II Tree length	0.0 ± 17.0	0.0 ± 5.0	4.0 ± 8.0
NSGA-II Visitation length	0.0 ± 2.0	0.0 ± 0.0	4.0 ± 1.0
NSGA-II Variables	97.0 ± 144.0	57.0 ± 140.0	44.5 ± 64.0

As argued previously, the solution length correlates with complexity, but the semantics have also be included to precisely evaluate the complexity of the generated solutions. Therefore, we analyzed which symbols acting as mathematical functions are present in the symbolic expression trees encoding the solutions and how many nodes are affected by them. Three groups of complex functions are analyzed. The group *trigonometric* functions includes $\sin$, $\cos$, and $\tan$, *exponential* functions includes $\exp$ and $\log$ and *power* functions includes n^2 and $\sqrt{n}$.

The analysis is based on the size of affected subtrees by symbols contained in the defined groups. For example, a possible solution $x_1 + \sin(x_2 + 5.0)$ represented as expression tree, contains four nodes that are affected by a trigonometric function. The sine itself, the addition and its two arguments the variable x_2 and the numerical constant 5.0. If multiple symbols contained in one of the defined groups are present in a solution, the number of affected nodes is summed up for all occurrences and thus this measure can exceed the solution length. The results of this symbol analysis reveal whether an algorithm could detect the appropriate symbols and thus mathematical functions to include in the symbolic expression trees.

The results of the symbol analysis for the 50 performed repetitions are shown in Table 4.5 as the median $\pm$ interquartile range of the affected nodes by each group. The optimal values for the number of affected nodes are written in bold font for each problem to have a reference value for comparison. This reference value has been calculated by using the data generating functions in Table 4.3. However, the number of affected nodes by power functions can fall below the optimal value, because the square functions can be represented by the multiplication of a term with itself, which gives a larger solution without any power functions at all.

The algorithm configurations, which exhibit the least pressure to generate parsimonious solutions, *GP Length 50*, *GP Length 100*, and *NSGA-II Variables* include lots of trigonometric, exponential, or power symbols without the actual need for that. Out of the single-objective standard genetic programming variants only *GP Length 20* can compete with the remaining multi-objective configurations. The reason therefore is that due to the tightly restricted solution length, every additional unnecessary symbol would results in a degradation of accuracy and is hence avoided.

With respect to the symbol analysis *NSGA-II Recursive*, *NSGA-II Tree length*, and *NSGA-II Visitation length* perform similarly. The largest difference between these algorithm variants is present on *Problem* F_1 and F_3, where an increased interquartile range can be observed when another complexity measure as the recursive one is used.

When comparing all obtained results regarding the accuracy, solution length, and symbol analysis the best algorithm variants are *GP Length 20* and NGSA-II with recursive complexity or visitation length. *GP Length 20* performs that well, because all of theses benchmark problems can be solved optimally within the limits enforced by the maximum tree length. However, when applied to real-world problems the concrete tree length to create accurate solution is not known a-priori and multiple configurations have to be tested. Contrary to this, the two NSGA-II variants adapt the solution length automatically and achieve as good results while being able to create larger models if necessary.

Exemplary Regression Models

The effects of performing multi-objective symbolic regression are illustrated by comparing the best models generated for problem F_2. The most accurate model of each algorithm is extracted from its 50 repetitions and its size statistics in terms of expression tree depth and length are listed in Table 4.6. In the *Original* column the results are stated for the solution as it is directly outputted by the algorithm. The results in the *Simplified* column are obtained by applying mathematical transformations and constant folding of the obtained solutions. These simplifications results in an decrease in solution length for all variants, but the most drastic improvement is obtained for the solution of *NSGA-II Variables*. This can be explained by the fact that this variant can build very large solutions as long as no additional variables are contained in the solutions and then constant folding is able to reduce the length of these solution significantly.

Table 4.6: Comparison of the most accurate solutions of each algorithm created with and without simplification.

Problem F_2	Original		Simplified		Equation
	Length	Depth	Length	Depth	
GP Length 20	18	7	10	4	Eq. 4.8
GP Length 50	39	11	20	6	Eq. 4.10
GP Length 100	64	21	54	15	Eq. 4.11
NSGA-II Recursive	16	8	9	4	Eq. 4.9
NSGA-II Tree length	16	7	10	4	Eq. 4.8
NSGA-II Visitation length	16	7	10	4	Eq. 4.8
NSGA-II Variables	77	22	10	4	Eq. 4.8

After simplification and manual adaption of coefficients by CO-NLS, all solutions describe the data accurately and equally well with a NMSE on training and test in the range of 10^{-20}. The simplified models of the NSGA-II configurations and *GP Length 20* are all exactly the same. This is not the case for the single-objective genetic programming algorithms with higher maximum tree length limits. As these solution have also a very high accuracy this is an indication for the presence of introns in the solutions, because otherwise the solution accuracy would be reduced.

Next to the size statistics the reference for the equation that represents the simplified solution as mathematical formula is given. There it can be seen that *GP Length 20* and the last three NSGA-II variants identified the data generating function exactly (Equation (4.8)). *NSGA-II Recursive* created an alternative formulation that is slightly shorter by singling out the variable x_4 (Equation (4.9)).

GP Length 50 and *GP Length 100* created bloated solutions, which contain additional terms in the mathematical formulas stated in Equation (4.10) and Equation (4.11) respectively. In Equation (4.10) the last term including the division of x_5 and x_1 just expresses 1.0 as the factor before the division is very small. The same principle applies to Equation (4.11), where the enormous term of nested trigonometric function is scaled by 8.710^{-7} and thus results in a very small number, which approximates zero. These bloated individuals containing deeply nested functions are avoided by either setting an appropriate maximum tree length or when switching from single-objective to multi-objective symbolic regression.

$$f_1(x) = x_2 + x_3x_4 + x_4x_5 + \sin(x_1) \tag{4.8}$$

$$f_2(x) = x_2 + x_4\,(x_3 + x_5) + \sin(x_1) \tag{4.9}$$

$$f_3(x) = x_2 + x_3x_4 + x_4x_5 + \sin(x_1)[5.11\ 10^{-10}\frac{x5}{x1} + 1] \tag{4.10}$$

$$\begin{aligned} f_4(x) = {} & x_2 + x_3x_4 + x_4x_5 + \sin(x_1) + 8.7\ 10^{-7}\cos(\cos(\sin(0.99x_1)) \\ & + e^{\sin(x_1)})) + 8.85\ 10^{-7}\cos(\sin(0.79\sin(x_1) + \sin(e^{\cos(1.61+e^{\sin(x_1)})}) \\ & \cdot \cos(\cos(\sin^2(\cos(\cos(\tan(\sin(0.99x_1)) + \cos(\sin(0.99x_1))))))))) \end{aligned} \tag{4.11}$$

4.3.3 Results on Noisy Problems

After initial tests on the newly defined benchmark problems, we performed additional tests with more difficult data sets. A description of those problems is given in Table 4.7. The 10-dimensional *Breiman* problem [Breiman et al., 1984] contains 5,000 training and test samples and the value of the input variables are sampled from the set $\{-1, 0, -1\}$, with the exception of x_1 that is sampled from $\{-1, 1\}$. The noise term ϵ accounts for 10% of the variance of the data generating function. The *Friedman* problem [Friedman, 1991] has the same number of training and test samples and each variable is sampled uniformly from $U[0, 1]$. The signal-to-noise ratio for this problem is 3.28, thus the noise term ϵ accounts for 9% of the total variance [Friedman, 1991]. Both described problems further test the feature selection capabilities of the algorithms, because not all input variables are actually used to calculate the response value.

The remaining three benchmark problems are real-world problems, where no mathematical functions has been used to generate the target values. The *Housing* problem [Harrison Jr and Rubinfeld, 1978] contains 13 input features that describe economical factors of the suburbs of Boston and the goal is to predict the median price value of owner-occupied homes. The *Chemical* problem [White et al., 2013] has been used previously in a symbolic regression competition and contains 57 measurements of a chemical production process. The *Tower* problem [Vladislavleva et al., 2009], also used for evaluating constants optimization, is related to the chemical one, because it origins from the same production process, but contains fewer input variables. Based on these benchmark problems that are more difficult to solve than the previously used ones, the performance of multi-objective symbolic regression is evaluated.

Table 4.7: Definition of noisy benchmark problems.

Name	Function	Training	Test
Breiman	$F_6(x_1, ..., x_{10}) = \begin{cases} 3 + 3x_2 + 2x_3 + x_4 + \epsilon & \text{if } x_1 = 1 \\ -3 + 3x_5 + 2x_6 + x_7 + \epsilon & \text{otherwise} \end{cases}$	5000 points	5000 points
Friedman	$F_7(x_1, ..., x_{10}) = 0.1e^{4x_1} + 4/[1 + e^{-20x_2+10}] + 3x_3 + 2x_4 + x_5 + \epsilon$	5000 points	5000 points
Housing	$F_8(x_1, ..., x_{13}) = ?$	337 points	169 points
Chemical	$F_9(x_1, ..., x_{57}) = ?$	711 points	355 points
Tower	$F_{10}(x_1, ..., x_{25}) = ?$	3136 points	1863 points

The difference to the problems used in the previous section is that these problem contain much more features to build regression models from. Furthermore, none of the noisy problems can be solved exactly with the configured algorithms, because all problems contain some noise and it not possible to generate a solution with no or almost zero estimation errors. Hence, the experiments with noisy problems simulate a more practical relevant setting.

The performance of the same algorithm variants described in Table 4.2 is evaluated on these noisy problems and again 50 repetitions of each configuration have been performed to take stochastic effects into account. The obtained results are presented in the same way as previously, by analyzing the most accurate solution for each algorithm execution. The results with respect to the accuracy of the solutions on training and test and the solution length are listed as median ± interquartile range in Table 4.8.

The first observation and difference to the previous experiment is that the worst performing algorithm across all noisy problems is *GP Length 20*, while it has been among the best when solving the artificial benchmark problems. The explanation therefore is that a maximum tree length of 20 does not suffice to create accurate regression models for these more complicated problems.

A similar phenomenon can be observed when studying the results obtained by *NSGA-II Variables*. In terms of the accuracy of the created solutions it is always among the worst performing multi-objective algorithms, except on the *Friedman* problem where no large differences can be detected. Again, the variables complexity measure applies no real parsimony pressure and although the variant performs competitively when compare to the single-objective variants, this can be mostly attributed to the NSGA-II algorithm.

When comparing the results obtained on the *Breiman* problem one notices the decrease in accuracy when increasing the maximum tree length from 50 to 100. The rationale is the same as before that due to the increased space of hypotheses the appropriate symbols and variables could not be identified anymore. However, here the impact of this effect is amplified. Among the NSGA-II variants the recursive complexity measure yielded the best performance on training and test and the smallest variance as well. Furthermore, this algorithm configuration creates the shortest solutions that still can explain the data very well. Solutions with a similar length are only obtained by *GP Length 50*, but their accuracy is much worse compared to the multi-objective algorithms.

The solutions created to the *Friedman* problem perform all pretty similarly, especially when evaluating their generalization abilities on the test partition. *NSGA-II Recursive* yielded the worst results in terms of accuracy on training and test, neglecting *GP Length 20* as it performs much worse.

This is especially remarkable, because for all other problems *NSGA-II Recursive* generates the best solutions. An explanation might be that data generating function can not be identified, because of its complex individual terms and the recursive complexity measure has been specifically designed to avoid those if possible.

On the *Housing* problem, all algorithms that can create solutions whose expression tree representation can exceed 50 nodes have a similar training performance. However, when comparing the test performance the solutions' accuracy gets worse, which is an indication of overfitting. Especially, when taken into account that *GP Length 20* exhibits one of the best test performances while creating pretty small solutions. The only algorithm capable to outperform *GP Length 20* is *NSGA-II Recursive*, which although it creates much larger solutions, tries to reduce their complexity, which is a possible explanation why overfitting happens to a lesser extend.

Solutions overfitting the training data are also created for the *Chemical* problem. The differences in the median training and test performances are between 0.06 and 0.16. The one outstanding algorithm on this problem is again *NSGA-II Recursive* that creates the most accurate solutions that generalize well on the test partition. Additionally, the median solution length is with 34.5 by far the smallest of the algorithms with a length limit of 100. The second best performing algorithm is NSGA-II using the visitation length as complexity measure, which falls only slightly behind NSGA-II with the recursive complexity measures. An interesting effect is that although *NSGA-II Visitation length* only minimizes the tree length and does not consider the complexity and semantics of solutions in any form, the median solution length is twice as high when compared to *NSGA-II Recursive*.

The solutions on the *Tower* problem do not vary much. The training performance is slightly below 0.12 with similar interquartile ranges and also the test performance stays around 0.125 for NSGA-II with recursive, tree length or visitation length as complexity measure. The only noteworthy observation on these results is that *NSGA-II* creates solutions with only half the size of the solution of other algorithms, while still maintaining a pretty high accuracy.

Further insights on these results are generated by performing the symbol analysis, how man nodes are affected by trigonometric, exponential, or power functions, on the created solutions. These results are again collected as the median ± interquartile range of the affected nodes for each defined group in Table 4.9.

Table 4.8: Training and test accuracy of the best training solution for each algorithm variant in terms of the normalized mean squared error and the solution length. All quantities are expresses as median ± interquartile range.

	Training	**Test**	**Length**
Breiman			
GP Length 20	0.263 ± 0.155	0.262 ± 0.159	16.5 ± 6
GP Length 50	0.181 ± 0.226	0.185 ± 0.213	39.5 ± 11
GP Length 100	0.560 ± 0.431	0.548 ± 0.452	85.0 ± 19
NSGA-II Recursive	0.105 ± 0.011	0.106 ± 0.011	39.5 ± 16
NSGA-II Tree length	0.112 ± 0.017	0.113 ± 0.018	56.5 ± 42
NSGA-II Visitation length	0.114 ± 0.025	0.114 ± 0.024	51.0 ± 43
NSGA-II Variables	0.138 ± 0.049	0.138 ± 0.050	75.5 ± 17
Friedman			
GP Length 20	0.193 ± 0.022	0.190 ± 0.022	17.0 ± 3
GP Length 50	0.140 ± 0.007	0.142 ± 0.005	47.0 ± 10
GP Length 100	0.141 ± 0.006	0.147 ± 0.007	85.0 ± 17
NSGA-II Recursive	0.147 ± 0.048	0.155 ± 0.045	45.0 ± 31
NSGA-II Tree length	0.135 ± 0.010	0.143 ± 0.012	57.5 ± 45
NSGA-II Visitation length	0.137 ± 0.010	0.146 ± 0.013	47.0 ± 43
NSGA-II Variables	0.132 ± 0.003	0.140 ± 0.003	78.0 ± 20
Housing			
GP Length 20	0.192 ± 0.014	0.198 ± 0.017	20.0 ± 3
GP Length 50	0.153 ± 0.017	0.211 ± 0.056	48.5 ± 7
GP Length 100	0.132 ± 0.023	0.202 ± 0.092	92.0 ± 12
NSGA-II Recursive	0.136 ± 0.028	0.176 ± 0.040	81.0 ± 24
NSGA-II Tree length	0.128 ± 0.027	0.231 ± 0.197	93.5 ± 24
NSGA-II Visitation length	0.125 ± 0.031	0.204 ± 0.051	87.0 ± 23
NSGA-II Variables	0.131 ± 0.028	0.215 ± 0.051	76.0 ± 12
Chemical			
GP Length 20	0.272 ± 0.021	0.432 ± 0.115	17.0 ± 4
GP Length 50	0.214 ± 0.026	0.329 ± 0.202	43.0 ± 11
GP Length 100	0.195 ± 0.026	0.343 ± 0.285	82.5 ± 18
NSGA-II Recursive	0.203 ± 0.017	0.263 ± 0.073	34.5 ± 36
NSGA-II Tree length	0.210 ± 0.032	0.302 ± 0.113	62.5 ± 30
NSGA-II Visitation length	0.205 ± 0.031	0.283 ± 0.127	67.0 ± 41
NSGA-II Variables	0.211 ± 0.040	0.337 ± 0.227	75.5 ± 15
Tower			
GP Length 20	0.158 ± 0.030	0.159 ± 0.034	18.0 ± 4
GP Length 50	0.138 ± 0.028	0.141 ± 0.035	43.0 ± 6
GP Length 100	0.124 ± 0.022	0.131 ± 0.028	89.0 ± 19
NSGA-II Recursive	0.124 ± 0.022	0.124 ± 0.023	34.5 ± 27
NSGA-II Tree length	0.120 ± 0.024	0.121 ± 0.026	77.5 ± 43
NSGA-II Visitation length	0.124 ± 0.023	0.125 ± 0.022	77.0 ± 41
NSGA-II Variables	0.133 ± 0.040	0.136 ± 0.040	76.0 ± 15

Table 4.9: Symbol analysis of the best solution in terms of the affected subtree length expressed as the median ± interquartile range.

	Trigonometric Functions	**Exponential Functions**	**Power Functions**
Breiman			
GP Length 20	0.0 ± 2.0	2.0 ± 6.0	0.0 ± 2.0
GP Length 50	18.5 ± 32.0	12.5 ± 21.0	9.0 ± 32.0
GP Length 100	110.0 ± 118.0	75.0 ± 58.0	31.5 ± 90.0
NSGA-II Recursive	0.0 ± 0.0	0.0 ± 2.0	0.0 ± 0.0
NSGA-II Tree length	0.0 ± 14.0	9.0 ± 23.0	0.0 ± 0.0
NSGA-II Visitation length	0.0 ± 5.0	7.0 ± 25.0	0.0 ± 0.0
NSGA-II Variables	165.0 ± 204.0	95.5 ± 124.0	43.0 ± 82.0
Friedman			
GP Length 20	4.0 ± 5.0	0.0 ± 2.0	3.0 ± 7.0
GP Length 50	40.0 ± 39.0	8.5 ± 20.0	8.0 ± 27.0
GP Length 100	105.0 ± 96.0	36.5 ± 54.0	37.0 ± 62.0
NSGA-II Recursive	11.0 ± 27.0	0.0 ± 2.0	2.0 ± 10.0
NSGA-II Tree length	46.5 ± 131.0	0.0 ± 7.0	11.0 ± 30.0
NSGA-II Visitation length	27.5 ± 47.0	0.0 ± 5.0	7.0 ± 24.0
NSGA-II Variables	222.0 ± 163.0	48.0 ± 88.0	68.0 ± 83.0
Housing			
GP Length 20	4.0 ± 7.0	4.0 ± 14.0	0.0 ± 6.0
GP Length 50	17.0 ± 19.0	26.0 ± 53.0	23.5 ± 34.0
GP Length 100	70.5 ± 86.0	98.0 ± 125.0	78.5 ± 136.0
NSGA-II Recursive	20.0 ± 30.0	13.5 ± 20.0	0.0 ± 4.0
NSGA-II Tree length	28.5 ± 53.0	66.5 ± 148.0	17.5 ± 74.0
NSGA-II Visitation length	14.0 ± 56.0	44.5 ± 109.0	9.0 ± 44.0
NSGA-II Variables	134.0 ± 201.0	117.5 ± 99.0	85.0 ± 75.0
Chemical			
GP Length 20	0.0 ± 2.0	0.0 ± 0.0	0.0 ± 6.0
GP Length 50	10.5 ± 24.0	0.0 ± 9.0	8.0 ± 15.0
GP Length 100	50.0 ± 86.0	18.0 ± 52.0	48.0 ± 56.0
NSGA-II Recursive	0.0 ± 4.0	0.0 ± 0.0	0.0 ± 11.0
NSGA-II Tree length	4.0 ± 36.0	0.0 ± 6.0	40.0 ± 69.0
NSGA-II Visitation length	21.5 ± 53.0	0.0 ± 0.0	10.0 ± 36.0
NSGA-II Variables	148.5 ± 216.0	41.0 ± 71.0	113.5 ± 99.0
Tower			
GP Length 20	0.0 ± 4.0	0.0 ± 0.0	0.0 ± 2.0
GP Length 50	14.0 ± 24.0	7.0 ± 19.0	7.5 ± 12.0
GP Length 100	56.0 ± 93.0	26.5 ± 84.0	37.0 ± 90.0
NSGA-II Recursive	0.0 ± 12.0	0.0 ± 2.0	0.0 ± 4.0
NSGA-II Tree length	37.0 ± 121.0	48.5 ± 113.0	3.0 ± 55.0
NSGA-II Visitation length	23.5 ± 57.0	20.0 ± 36.0	4.0 ± 22.0
NSGA-II Variables	270.0 ± 275.0	96.0 ± 95.0	72.5 ± 121.0

For solving the *Breiman* problem no trigonometric, exponential, or power functions are necessary. However, this is only detected by *GP Length 20* and *NSGA-II Recursive* and to a lesser extend by the other multi-objective variants using the tree or visitation length. The single-objective algorithms with larger tree limits and NSGA-II with the variables complexity measure use functions of these groups rather liberally, which explains there worse accuracy on this problem.

The *Friedman* problem can be solved by using only arithmetic and exponential functions. Curiously, all algorithms include trigonometric functions, though these are unnecessary. Only *GP Length 100* and *NSGA-II* include the necessary exponential functions, whereas the other NSGA-II configurations do not include these functions at all. However, this is not reflected in the accuracy of the generated solutions. Therefore, it can be concluded that none of the algorithms is able to identify the data generating function without noise accurately.

While observing the quality results on the *Housing* problem we noticed that the generated solutions overfit the training data. This is also visible in the symbol analysis, because *NSGA-II Recursive* that performed significantly better on test than the other algorithms, uses the least complex functions indicated by low median values and even lower interquartile ranges. The only algorithm that can compete with that is *GP Length 20*, which solutions perform slightly worse on the test partition.

Overfitting also occurs when building solutions for the *Chemical* problem. The least overfit solutions have been created by *NSGA-II Recursive*, which also uses the fewest complex mathematical functions to create the solutions. Therefore, it can be concluded that the recursive complexity measure gives the algorithm an advantage to asses the real complexity of solutions by the inclusion of symbol semantics, especially when compared to using just the shape or length of the symbolic expression tree as complexity measure.

The solutions to the *Tower* problem have similar characteristics as solutions to the *Chemical* problem. The most compact and accurate solutions have been created by *NSGA-II Recursive*, which is also indicated by the symbol analysis. Almost no trigonometric, exponential, or power symbols are included in the obtained solutions when using the recursive complexity measure. This is not the case for any other algorithm, except *GP Length 20* that generates not as accurate solutions.

In summary it can be concluded that, when considering both benchmark problem suites, multi-objective symbolic regression with the recursive complexity measure performs best of all algorithm variants. This highlights the suitability of the recursive complexity measure for performing multi-objective symbolic regression.

Exemplary solutions to the Tower problem

The effects of switching from single-objective to multi-objective symbolic regression are revealed by studying the algorithm dynamics during the optimization. Therefore, we selected the most accurate solutions obtained by single-objective and multi-objective symbolic regression. The best solution created by a single-objective algorithm (*GP Length 100*) consists of 91 expression tree nodes and the mean absolute error (MAE) is 20.80 on the training partition and 19.76 on the test partition (NMSE training 0.095, NMSE test 0.097). The most accurate multi-objective solution was created by *NSGA-II Recursive* and consists of 99 expression tree nodes. It is a little more accurate with a MAE of 20.23 on training and 18.80 on test (NMSE training 0.097, NMSE test 0.088).

These quantities are calculated directly on the algorithm output. When performing constants folding, mathematical transformation and simplifications, and constants optimization by CO-NLS the results of the standard genetic programming solution can only be slightly improved. The improved solution consists of 85 tree nodes and has a training MAE of 20.08 and a test NMSE of 19.76. In contrast to this, the multi-objective solutions gets reduced to 46 tree nodes by the applied simplifications and CO-NLS reduces the training MAE to 19.09 and the test MAE to 17.50.

More interesting than the qualities and size statistics is how these solutions have been generated and what symbols are used during the optimization. Therefore, the relative number of symbol occurrences over all solutions in the population have been tracked during the algorithm execution for each generation.

In Figure 4.9 the relative symbol frequencies of the *GP Length 100* algorithm that created the discussed solution are plotted. The symbols have been group similarly as before to increase the clarity of the chart. At the start all symbol occur with the same probability and after that more important ones appear more often. Until the algorithm stops several exponential or power symbol and trigonometric symbols are used, although these are not necessary to create accurate solutions.

This picture changes completely when switching to multi-objective symbolic regression. The relative variable frequencies for the *NSGA-II Recursive* algorithm are shown in Figure 4.10. There, all exponential, power, and trigonometric symbols are removed within the first few generations and only arithmetic or terminal symbols remain in the solutions. This is the result of the algorithm identifying appropriate symbols for modeling the data and the inclusion of semantics in the recursive complexity measure.

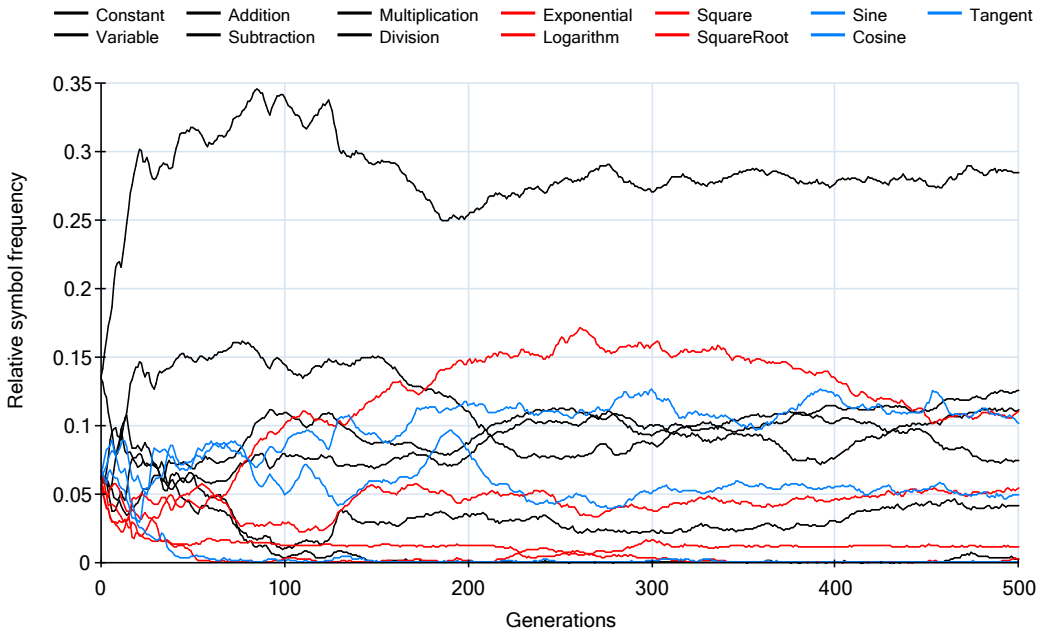


Figure 4.9: Symbol frequencies of *GP Length 100* execution.

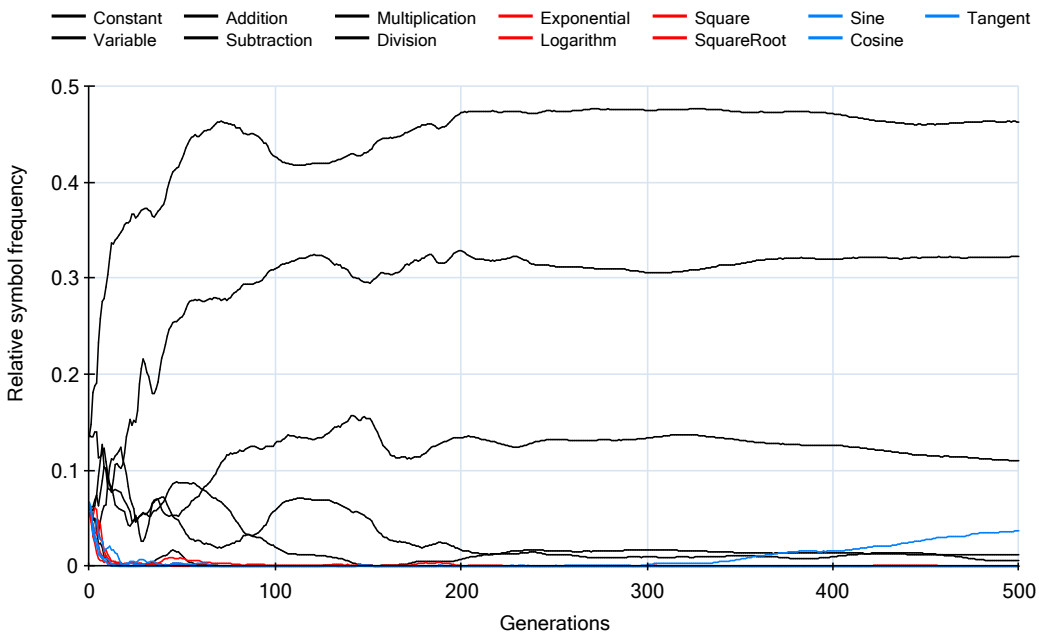


Figure 4.10: Symbol frequencies of *NSGA-II Recursive* execution.

4.4 Concluding Remarks

In this chapter the complexity of symbolic regression models and multi-objective symbolic regression have been discussed. The complexity of regression models is critical, because the less complex yet accurate a solution is, the better is its interpretability, which is a key argument for performing symbolic regression. Therefore, several complexity measures proposed in the literature have been reviewed. Based on experience from performing symbolic regression for real-world data modeling and desired characteristics a new complexity measure has been defined.

The recursive complexity takes, in contrast to other complexity measures, the semantics of the symbols included in the symbolic expression trees into account. Additionally, it can be efficiently calculated within one tree traversal and does not require any evaluation of the solution. A drawback of the recursive complexity is that its value is not meaningful, because it has been specifically designed for the use as an additional objective in symbolic regression and hence its value is hard to interpret.

Afterwards, different multi-objective symbolic regression approaches have been reviewed and the nondominated sorting genetic algorithm (NSGA-II) is discussed. Initial test for evaluating its applicability to solve multi-objective symbolic regression problems were disappointing and to increase its performance the algorithm had to be adapted. These adaptations include a relaxed domination criterion and discretized objective functions.

The goal of switching from single-objective to multi-objective symbolic regression is that the maximum tree length and the appropriate function set do not have to be specified a-priori, but are automatically detected by the algorithm. To test if this claim holds, we created five artificial benchmark problems and evaluated the performance of three different single-objective genetic programming configurations with varying maximum tree size. Furthermore, four different variants of the multi-objective NSGA-II algorithm, with different complexity measures used as secondary objective next to the solution accuracy, are evaluated.

The results obtained by NSGA-II with the recursive complexity measures are among the best when solving the artificial benchmark problems, but the differences among the individual variants are rather small. Therefore, we evaluated all seven algorithm variants on problems that cannot be solved exactly, due to noise contained in the data, and again NSGA-II with recursive complexity yielded the most accurate and compact results.

Additionally, symbol analysis has been performed that reveals which symbols are included in the solutions and whether an algorithm can identify the appropriate symbols for solving a problem. Whereas all single-objective al-

gorithm use more complex symbols such as trigonometric or exponential ones rather liberally, multi-objective algorithms can detect the necessary functions to model the data rather accurately.

Although the recursive complexity might not provide the most accurate estimation of a solution's complexity, it compromises evaluation speed and accuracy. When used for solving multi-objective symbolic regression problems the algorithm performance can be improved across all tested problems. Furthermore, it can at least compete or even outperform single-objective genetic programming for solving symbolic regression problems. Therefore, it can be concluded that multi-objective symbolic regression, especially when the recursive complexity measure is used as objective, is able to generate simple and accurate solutions for symbolic regression problems without the need for specifying tree limits or the appropriate function set.

Chapter 5

Conclusion

The main topic of this thesis is symbolic regression and how algorithms performing symbolic regression can be improved. The outstanding characteristics of symbolic regression is that models are created in the form of mathematical expressions without any assumptions about the model structure. However, this advantage complicates the generation of highly accurate yet still interpretable regression models and the number of possible models describing the data is basically endless. Therefore, genetic programming or variants thereof are commonly used for solving symbolic regression problems.

The main contributions in this work to improve the state of the art are methods for local optimization and complexity control for symbolic regression. Local optimization refers in this context to a new way for the identification of numerical model parameters. After a model is created its parameters are adapted by applying damped least squares optimization by the Levenberg-Marquardt algorithm. The Levenberg-Marquardt algorithm is an iterative optimization method that utilizes the gradient information that is in this case provided by automatic differentiation of the symbolic regression model according to its numeric parameters. Although, this local optimization method termed constants optimization by nonlinear least squares (CO-NLS) is computationally expensive, it improves the accuracy of the generated models significantly. Furthermore, it separates the problem in the identification of an appropriate model structure and the identification of appropriate numeric parameters, which is later performed by CO-NLS.

The benefits of symbolic regression, most importantly that the model are created in the form of interpretable mathematical expression, can only be utilized if the models are as simple and parsimonious as possible while still describing the data accurately. Therefore, methods for complexity control in symbolic regression have been investigated and a new complexity metric has been implemented. This recursive complexity is efficiently generated and

combines syntactical and semantical information about the models. This enables the use of multi-objective genetic programming for symbolic regression, where while the accuracy is maximized the model complexity is simultaneously minimized. The nondominated sorting genetic algorithm II (NSGA-II) has been adapted to solve multi-objective symbolic regression problems and its competitiveness with single-objective algorithm is demonstrated on several benchmark problems. Switching from single-objective to multi-objective symbolic regression alleviates the need to define restrictions on length of the models and the functions that can occur in the models, because the algorithm is capable of identifying appropriate values during the optimization.

In the following an overview of the main contributions and contents of this thesis is given.

Symbolic Regression

- The problem of symbolic regression is introduced and its advantages and disadvantages are highlighted. Afterwards, the main properties of genetic programming are detailed, because it is the most common method for solving symbolic regression problems. Genetic programming is an evolutionary meta-heuristic algorithm that works with a variable length representation, which is highly suitable for representing mathematical expression. The main concept of the algorithm is iterative recombination of high-quality solutions, which leads to even better solutions. After this general introduction it is shown how genetic programming can be used for solving symbolic regression problems in detail.

- Following this line of thought, frameworks and software for performing genetic programming and symbolic regression are presented. A framework significantly reduces the effort for implementing common tasks and allows the user to focus on developing new methods. All of the presented methods of this thesis are publicly available and implemented in the open-source framework for heuristic optimization *HeuristicLab*.

- Heuristic methods allow the exploration of an enormous search space in reasonable time. However, a drawback of heuristics is that due to their stochastic nature every algorithm execution yields different results. Therefore, fast function extraction (FFX) and prioritized grammar enumeration (PGE) are described as an alternative to genetic programming for solving symbolic regression problems.

FFX works by generating hundreds or even thousands basis functions that are derived from the input features. These basis functions are combined in a linear model that is created with elastic net regularization. FFX is amazingly fast due to the restriction to create generalized linear models from basis function. This restriction comes with a price, namely that more complicated nonlinear models cannot be created by FFX. Furthermore, it is quite common that a model generated by FFX contains more than 50 basis functions, which hampers the interpretability of the model.

PGE starts with a minimal set of basis functions and generates models from these. A grammar further defines how models can be extended in a structured way and every created model is stored in a Pareto priority queue. The numerical parameters of a model are estimated similarly to CO-NLS by the Levenberg-Marquardt algorithm. PGE generates models deterministically from simple to more complex ones and it can generate highly nonlinear models. A drawback however is that the number of possible models increases exponentially with the number of features.

Local Optimization

- Fitting the numerical parameters of models in symbolic regression is an important topic. The best model structure will not produce accurate predictions as long as its numerical parameters are not adapted to the problem at hand. A first approach to improve the numerical parameters is linear scaling, which removes the necessity of identifying the scale and the offset of the models. As a result the maximal prediction error of the scaled model is equal to the variance of the target values.

- Based on the success of the inclusion of linear scaling in symbolic regression, several other local optimization techniques that adapt the numerical coefficients of models have been created. The most intriguing methods include machine learning techniques for function fitting in symbolic regression, but were still outperformed by linear scaling.

- We combined linear scaling and with nonlinear gradient-based optimization techniques for the identification of numerical model parameter in CO-NLS. Therefore, the gradient of a model is created by automated differentiation that allows a fast and accurate calculation. CO-NLS is able to improve the accuracy of symbolic regression models within a few iterations of the algorithm.

- CO-NLS is applicable in genetic programming for symbolic regression, but the general methodology can be used for parameter tuning of any mathematical model structure for which the gradient can be calculated. Therefore, CO-NLS is additionally available as post-processing step to further increase the accuracy of generated models.

- The performance of CO-NLS is first evaluated on five benchmark problems that have been used for demonstrating the effectiveness of linear scaling and CO-NLS excels at solving these. However, only training performances are given for these problems in the original publication and when the generalization capabilities of the models are tested, none of the generated models, regardless of whether linear scaling or CO-NLS has been used during model creation, can explain the test data.

- As a result more balanced benchmark problems, with respect to training and test data, have been used for studying the effects of CO-NLS. We have studied the effect of the probability to apply CO-NLS to a solution and the number of iterations of CO-NLS that are performed. This investigations reveals that the higher both of these parameters are set, hence more local optimization is applied, the better the results are. However, not all benchmark problems can be solved and the runtime of the algorithm is increased due to the additional evaluations performed by CO-NLS.

- This same overall picture holds when comparing the different algorithm variants of offspring selection genetic programming. Again the higher the probability and iterations of CO-NLS the better the results. However, the baseline to compare against is higher, because of the additional offspring selection step, which discards unfit solution automatically. Offspring selection with CO-NLS yielded high success rates and even the most difficult benchmark problems could be solved.

  Based on these experiments we can conclude that CO-NLS improves the performance of the algorithms solving symbolic regression problems and enables them to create more accurate solutions.

Complexity Control

- Another important aspect of symbolic regression is the complexity of the generated solutions, because this directly correlates with their interpretability. Therefore, different complexity measures for symbolic regression are reviewed and compared.

- From this review of existing complexity measures, the recursive complexity is derived. The recursive complexity is named after its recursive definition, which aggregates the complexity of the child nodes until the root node of the symbolic regression model is reached. The advantage of this new complexity measure is that while it is efficiently calculated, it still incorporates the semantics of the encountered functions during the recursion. Therefore, it gives a better indication of the real complexity as syntactical complexity measures, which ignore the semantics completely, and is still faster to compute than complexity measures that require the model to be evaluated.

- The different complexity measures allow to switch from single-objective to multi-objective symbolic regression, where the accuracy and the complexity are the objectives to be optimized. Therefore, standard genetic programming is not applicable anymore and the NSGA-II is used. The original definition has to be adapted to the specifics of performing symbolic regression. The relaxed domination criterion ensures that the generated Pareto front does not consist solely of overly simplistic models containing only one input variable, thus hampering the optimization progress. The discretization of the objective functions further trims the Pareto front by assigning the same objective values to more solutions, which are in turn dominated by each other.

- Next we compared the performance of multi-objective symbolic regression with various complexity measures to single-objective symbolic regression performed by standard genetic programming. Whereas standard genetic programming has been performed with three different maximum tree sizes, the NSGA-II executions were only limited by the largest maximum tree size, because the expectation of multi-objective symbolic regression is that the appropriate tree size is automatically determined and it is not, or at least to a lesser extend than single-objective genetic programming, affected by bloat. Furthermore, an extended function set including trigonometric, exponential and power functions has been utilized in these experiments.

- The best performing single-objective algorithm on the first set of benchmark problems is the configuration with the strictest size limits. The reason therefore is that due to the tight restriction of the search space only parsimonious solutions can be created. The configurations with larger tree size perform worse when the generalization capabilities of the solutions are evaluated on the test partition and sometimes even the training performance is worse.

- The performances of the different multi-objective algorithm configurations have been similarly and all problems have been solved with very high accuracy. Only minor differences in favor of the recursive complexity and visitation length have been observed. The generated models are only slightly larger than those generated by standard genetic programming with the smallest tree size limits, although their configuration allows to build models that are five times as large.

- Afterwards the performance on noisy and real-world benchmark problems has been investigated. There, the single-objective configuration creating the smallest solutions does not perform well, because the solutions are too simplistic to describe the data adequately. The best performance over all noisy benchmark problems has been achieved by NSGA-II using the recursive complexity measure. This configuration created the simplest yet very accurate solutions.

- In addition, a symbol analysis of all the created models has been performed to evaluate, which mathematical functions have been integrated in the solutions and how they have been integrated. This is especially interesting when the data generating function is known, as it is often the case for benchmark problems. The fewest trigonometric, exponential, and power functions have been incorporated in the models by the single-objective algorithm with the smallest tree size and the multi-objective algorithm using the recursive complexity. The reasoning therefore is the same as before; the single-objective variant works that well because of the enormous parsimony pressure applied to the solutions. The benefit of the recursive complexity is that it integrates the semantics in the complexity calculation and hence implicitly minimizes the use of more complicated functions.

  The use of the NSGA-II for multi-objective symbolic regression alleviates the need to restrict the length of the models, reduces the occurrence of bloat, and if a complexity measure, which includes the semantic of the model is used, allows the algorithm to automatically identify appropriate mathematical functions. Furthermore, instead of single solution, the algorithm produces a Pareto front of solutions, which represents the tradeoff between accuracy and complexity when performing regression analysis.

This thesis claims by no means to have answered all the open questions to improve symbolic regression and even in the thematic focus of local optimization and complexity control there are several ways to further improve the methodology.

Future Research

- CO-NLS works by applying the Levenberg-Marquardt algorithm in combination with automatic differentiation for tuning numerical model parameters. However, based on the starting points for the gradient descent algorithm the nearest local optima for the numerical parameters is reached. The question remains if multiple restarts from varying starting points would further improve the achieved accuracy
- Another idea for future research is to evaluate different algorithms for local optimization instead of the Levenberg-Marquardt algorithm. For example, methods that are successfully applied in other machine learning techniques such as stochastic gradient descent and its extensions or mini batch training might be an appropriate choice .
- Furthermore, CO-NLS is currently computational expensive and its execution time scales directly with the number of training samples. Although it pays off to spend time for local optimization instead of evaluating different solutions, a more efficient version is desirable. Possible improvements regarding the computational efficiency range from distributing the effort over multiple computation units, to applying CO-NLS only to selected individuals, or reducing the number of parameters that are tuned by CO-NLS before it is applied.
- With respect to complexity control, the current approach of using NSGA-II with the accuracy and the recursive complexity as objectives works reasonable well and improves the overall performance of symbolic regression. However, the concrete definition of the recursive complexity is based on estimating appropriate rules to aggregate the individual complexity rules and desired properties of solutions. The question for further research is, if the aggregation rules can be adapted to include a-priori knowledge about the problem at hand and if that would further increase the performance of the algorithm and lead to even better and simpler solutions.
- The combination of CO-NLS and the complexity control method should further improve the accuracy of the generated models as it is the case for single-objective symbolic regression.

- In recent years age-layered population structures (ALPS) and age-fitness Pareto optimization gained popularity, as these techniques allow endless optimization and symbolic regression modeling on data streams. Especially, the Pareto optimization by NSGA-II can be easily extended to optimize the three objectives, model accuracy, complexity and age, thus enabling the algorithm to permanently create completely new solutions.

- The focus of creating methods for complexity control was to generate simpler regression models. The occurrence of overfitting is connected to the complexity of the generated models. Although, overfitting prevention has not been the motivation for creating the recursive complexity measure, the performed experiments already indicate that overfitting is reduced. However, to draw precise conclusions a detailed analysis on overfitting and complexity control in symbolic regression has to be performed.

- Symbolic regression is generally computationally expensive, because of the free-form models with little restriction that are created. As in most regression techniques the execution time scales at the very least linearly (often squared or with higher powers) with the amount of training samples. Therefore, sampling, clustering, or co-evolutionary techniques to reduce the training times and their effect on the algorithm have to be studied to enable the application of symbolic regression in scenarios, where lots of data is available.

- Deterministic or more robust symbolic regression algorithms are essential to gain acceptance of practitioners and the spread the usage of symbolic regression for data analysis tasks. CO-NLS and complexity control with the recursive complexity measure could build the foundations of new algorithms for symbolic regression. CO-NLS tackles the problem of identifying numeric parameters of the models and the recursive complexity structures the space of possible models, which allows a more directed exploration of the hypothesis space.

Bibliography

Michael Affenzeller and Stefan Wagner. Offspring selection: A new self-adaptive selection scheme for genetic algorithms. In B. Ribeiro, R. F. Albrecht, A. Dobnikar, D. W. Pearson, and N. C. Steele, editors, *Adaptive and Natural Computing Algorithms*, Springer Computer Science, pages 218–221. Springer, 2005.

Michael Affenzeller, Stephan Winkler, Stefan Wagner, and Andreas Beham. *Genetic Algorithms and Genetic Programming - Modern Concepts and Practical Applications*, volume 6 of *Numerical Insights*. CRC Press, Chapman & Hall, 2009.

Michael Affenzeller, Stephan Winkler, Gabriel Kronberger, Michael Kommenda, Bogdan Burlacu, and Stefan Wagner. Gaining deeper insights in symbolic regression. In Rick Riolo, Jason H. Moore, and Mark Kotanchek, editors, *Genetic Programming Theory and Practice XI*, Genetic and Evolutionary Computation. Springer, 2014.

Cesar L. Alonso, Jose Luis Montana, and Cruz Enrique Borges. Evolution strategies for constants optimization in genetic programming. In *21st International Conference on Tools with Artificial Intelligence, ICTAI '09*, pages 703–707, November 2009. doi: doi:10.1109/ICTAI.2009.35.

Peter J Angeline. Subtree crossover: Building block engine or macromutation. *Genetic Programming*, 97:9–17, 1997.

Peter J Angeline and Jordan Pollack. Evolutionary module acquisition. In *Proceedings of the second annual conference on evolutionary programming*, pages 154–163. Citeseer, 1993.

Ignacio Arnaldo, Krzysztof Krawiec, and Una-May O'Reilly. Multiple regression genetic programming. In *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation*, pages 879–886. ACM, 2014.

James Edward Baker. Adaptive selection methods for genetic algorithms. In *Proceedings of an International Conference on Genetic Algorithms and their applications*, pages 101–111. Hillsdale, New Jersey, 1985.

Andreas Beham, Johannes Karder, Gabriel Kronberger, Stefan Wagner, Michael Kommenda, and Andreas Scheibenpflug. Scripting and framework integration in heuristic optimization environments. In *Proceedings of the Companion Publication of the 2014 Annual Conference on Genetic and Evolutionary Computation*, pages 1109–1116. ACM, 2014.

Hans-Georg Beyer and Hans-Paul Schwefel. Evolution strategies–a comprehensive introduction. *Natural computing*, 1(1):3–52, 2002.

Åke Björck. *Numerical methods for least squares problems*. SIAM, 1996.

Tobias Blickle and Lothar Thiele. A comparison of selection schemes used in evolutionary algorithms. *Evolutionary Computation*, 4(4):361–394, 1996.

Markus Brameier. *On linear genetic programming*. PhD thesis, Universität Dortmund, 2005.

Leo Breiman, Jerome Friedman, Charles J Stone, and Richard A Olshen. *Classification and regression trees*. CRC press, 1984.

Anne Brindle. *Genetic algorithms for function optimization*. PhD thesis, University of Alberta, 1980.

Mauro Castelli, Leonardo Trujillo, Leonardo Vanneschi, Sara Silva, et al. Geometric semantic genetic programming with local search. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, pages 999–1006. ACM, 2015.

Qi Chen, Bing Xue, and Mengjie Zhang. Generalisation and domain adaptation in gp with gradient descent for symbolic regression. In *Evolutionary Computation (CEC), 2015 IEEE Congress on*, pages 1137–1144. IEEE, 2015.

Qi Chen, Mengjie Zhang, and Bing Xue. Feature selection to improve generalisation of genetic programming for high-dimensional symbolic regression. *IEEE Transactions on Evolutionary Computation*, 2017.

Xianshun Chen, Yew-Soon Ong, Meng-Hiot Lim, and Kay Chen Tan. A multi-facet survey on memetic computation. *IEEE Transactions on Evolutionary Computation*, 15(5):591–607, 2011.

Thomas F Coleman and Yuying Li. An interior trust region approach for nonlinear minimization subject to bounds. *SIAM Journal on optimization*, 6(2):418–445, 1996.

Vinícius Veloso de Melo, Benjamin Fowler, and Wolfgang Banzhaf. Evaluating methods for constant optimization of symbolic regression benchmark problems. In *Intelligent Systems (BRACIS), 2015 Brazilian Conference on*, pages 25–30. IEEE, 2015.

Kalyanmoy Deb and Himanshu Jain. An evolutionary many-objective optimization algorithm using reference-point-based nondominated sorting approach, part I: Solving problems with box constraints. *IEEE Transactions on Evolutionary Computation*, 18(4):577–601, 2014.

Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, 2002.

Kenneth DeJong. An analysis of the behavior of a class of genetic adaptive systems. *Ph. D. Thesis, University of Michigan*, 1975.

Stephen Dignum and Riccardo Poli. Operator equalisation and bloat free GP. In *Genetic Programming*, pages 110–121. Springer, 2008.

Norman Richard Draper, Harry Smith, and Elizabeth Pownell. *Applied regression analysis*, volume 3. Wiley New York, 1966.

Stuart Dreyfus. The numerical solution of variational problems. *Journal of Mathematical Analysis and Applications*, 5(1):30–45, 1962.

Bradley Efron, Trevor Hastie, Iain Johnstone, Robert Tibshirani, et al. Least angle regression. *The Annals of statistics*, 32(2):407–499, 2004.

David Elliott. The evaluation and estimation of the coefficients in the Chebyshev series expansion of a function. *Mathematics of Computation*, 18(86): 274–284, 1964.

Achiya Elyasaf and Moshe Sipper. Software review: the HeuristicLab framework. *Genetic Programming and Evolvable Machines*, 15(2):215–218, 2014.

Thomas Fernandez and Matthew Evett. Numeric mutation as an improvement to symbolic regression in genetic programming. In *Evolutionary Programming VII*, pages 251–260. Springer, 1998.

Cândida Ferreira. *Gene expression programming: mathematical modeling by an artificial intelligence*, volume 21. Springer, 2006.

R. Fletcher. *Practical Methods of Optimization; (2Nd Ed.)*. Wiley-Interscience, New York, NY, USA, 1987. ISBN 0-471-91547-5.

Félix-Antoine Fortin, François-Michel De Rainville, Marc-André Gardner, Marc Parizeau, and Christian Gagné. DEAP: Evolutionary algorithms made easy. *Journal of Machine Learning Research*, 13(Jul):2171–2175, 2012.

Jerome H Friedman. Multivariate adaptive regression splines. *The annals of statistics*, pages 1–67, 1991.

Christian Gagne and Marc Parizeau. Open BEAGLE: A new versatile C++ framework for evolutionary computations. In *Late-Breaking Papers of the 2002 Genetic and Evolutionary Computation Conference (GECCO 2002)*, pages 161–168, 2002.

Marc-André Gardner, Christian Gagné, and Marc Parizeau. Bloat control in genetic programming with a histogram-based accept-reject method. In *Proceedings of the 13th annual conference companion on Genetic and evolutionary computation*, pages 187–188. ACM, 2011.

David E Goldberg and Kalyanmoy Deb. A comparative analysis of selection schemes used in genetic algorithms. *Foundations of genetic algorithms*, 1: 69–93, 1991.

David E Goldberg et al. *Genetic algorithms in search optimization and machine learning*, volume 412. Addison-wesley Reading Menlo Park, 1989.

Andreas Griewank and Andrea Walther. *Evaluating derivatives: principles and techniques of algorithmic differentiation*. SIAM, 2008.

Steven Gustafson, Edmund K Burke, and Natalio Krasnogor. On improving genetic programming for symbolic regression. In *2005 IEEE Congress on Evolutionary Computation*, volume 1, pages 912–919. IEEE, 2005.

Isabelle Guyon and André Elisseeff. An introduction to variable and feature selection. *Journal of machine learning research*, 3(Mar):1157–1182, 2003.

Nikolaus Hansen, Sibylle D Müller, and Petros Koumoutsakos. Reducing the time complexity of the derandomized evolution strategy with covariance matrix adaptation (CMA-ES). *Evolutionary computation*, 11(1):1–18, 2003.

Kim Harries and Peter Smith. Exploring alternative operators and search strategies in genetic programming. *Genetic Programming*, 97:147–155, 1997.

David Harrison Jr and Daniel L Rubinfeld. Hedonic housing prices and the demand for clean air. *Journal of environmental economics and management*, 5(1):81–102, 1978.

Simon Haykin. *Neural Networks: A Comprehensive Foundation*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2nd edition, 1998. ISBN 0132733501.

Magnus Rudolph Hestenes and Eduard Stiefel. Methods of conjugate gradients for solving linear systems. *Journal of Research of the National Bureau of Standards*, 49(6):409–436, December 1952.

Arthur E Hoerl and Robert W Kennard. Ridge regression: Biased estimation for nonorthogonal problems. *Technometrics*, 12(1):55–67, 1970.

John Henry Holland. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. MIT press, 1992.

Gregory S Hornby. ALPS: the age-layered population structure for reducing the problem of premature convergence. In *Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 815–822. ACM, 2006.

Les M. Howard and Donna J. D'Angelo. The GA-P: A genetic algorithm and genetic programming hybrid. *IEEE Expert*, 10(3):11–15, 1995.

Himanshu Jain and Kalyanmoy Deb. An evolutionary many-objective optimization algorithm using reference-point based nondominated sorting approach, part II: Handling constraints and extending to an adaptive approach. *IEEE Trans. Evolutionary Computation*, 18(4):602–622, 2014.

Eric Jones, Travis Oliphant, Pearu Peterson, et al. SciPy: Open source scientific tools for Python, 2001–. URL http://www.scipy.org/.

Perla Juárez-Smith and Leonardo Trujillo. Integrating local search within neat-GP. In *Proceedings of the 2016 Genetic and Evolutionary Computation Conference Companion*, pages 993–996. ACM, 2016.

Maarten Keijzer. Improving symbolic regression with interval arithmetic and linear scaling. In Conor Ryan, Terence Soule, Maarten Keijzer, Edward Tsang, Riccardo Poli, and Ernesto Costa, editors, *Proceedings of the 6th European Conference on Genetic Programming, EuroGP 2003*, volume 2610 of *LNCS*, pages 70–82, Essex, 2003. Springer Berlin Heidelberg.

Maarten Keijzer. Scaled symbolic regression. *Genetic Programming and Evolvable Machines*, 5(3):259–269, 2004. ISSN 1389-2576.

Maarten Keijzer and James Foster. Crossover bias in genetic programming. In *Genetic Programming*, pages 33–44. Springer, 2007.

Kenneth E Kinnear Jr. Generality and difficulty in genetic programming: Evolving a sort. In *ICGA*, pages 287–294, 1993.

Scott Kirkpatrick, C Daniel Gelatt, Mario P Vecchi, et al. Optimization by simulated annealing. *science*, 220(4598):671–680, 1983.

Michael Kommenda, Gabriel Kronberger, Stefan Wagner, Stephan Winkler, and Michael Affenzeller. On the architecture and implementation of tree-based genetic programming in heuristiclab. In *Proceedings of the 14th annual conference companion on Genetic and evolutionary computation*, pages 101–108. ACM, 2012.

Michael Kommenda, Michael Affenzeller, Gabriel Kronberger, and Stephan M Winkler. Nonlinear least squares optimization of constants in symbolic regression. In *International Conference on Computer Aided Systems Theory*, pages 420–427. Springer, 2013a.

Michael Kommenda, Gabriel Kronberger, Stephan Winkler, Michael Affenzeller, and Stefan Wagner. 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*, pages 1121–1128. ACM, 2013b.

Michael Kommenda, Andreas Beham, Michael Affenzeller, and Gabriel Kronberger. Complexity measures for multi-objective symbolic regression. In *International Conference on Computer Aided Systems Theory*, pages 409–416. Springer, 2015.

Michael Kommenda, Gabriel Kronberger, Michael Affenzeller, Stephan M Winkler, and Bogdan Burlacu. Evolving simple symbolic regression models by multi-objective genetic programming. In *Genetic Programming Theory and Practice XIII*, pages 1–19. Springer, 2016.

Mark Kotanchek, Guido Smits, and Ekaterina Vladislavleva. Pursuing the pareto paradigm: Tournaments, algorithm variations and ordinal optimization. In *Genetic Programming Theory and Practice IV*, pages 167–185. Springer, 2007.

Mark Kotanchek, Guido Smits, and Ekaterina Vladislavleva. Trustable symbolic regression models: using ensembles, interval arithmetic and pareto fronts to develop robust and trust-aware models. *Genetic programming theory and practice V*, pages 201–220, 2008.

John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA, 1992.

John R. Koza. *Genetic programming II: automatic discovery of reusable programs*. MIT Press, Cambridge, MA, USA, 1994. ISBN 0-262-11189-6.

Krzysztof Krawiec. Genetic programming with local improvement for visual learning from examples. In *Computer Analysis of Images and Patterns*, pages 209–216. Springer, 2001.

Gabriel Kronberger. *Symbolic Regression for Knowledge Discovery – Bloat, Overfitting, and Variable Interaction Networks*. Number 64 in Johannes Kepler University, Linz, Reihe C. Trauner Verlag+Buchservice GmbH, 2011.

Gabriel Kronberger, Stephan Winkler, Michael Affenzeller, and Stefan Wagner. On crossover success rate in genetic programming with offspring selection. In *European Conference on Genetic Programming*, pages 232–243. Springer, 2009.

William La Cava and Jason Moore. A general feature engineering wrapper for machine learning using ϵ-lexicase survival. In *European Conference on Genetic Programming*, pages 80–95. Springer, 2017.

William La Cava, Thomas Helmuth, Lee Spector, and Kourosh Danai. Genetic programming with epigenetic local search. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, pages 1055–1062. ACM, 2015.

Fergal Lane, R Muhammad Atif Azad, and Conor Ryan. On effective and inexpensive local search techniques in genetic programming regression. In *International Conference on Parallel Problem Solving from Nature*, pages 444–453. Springer, 2014.

William B. Langdon. Size fair and homologous tree crossovers for tree genetic programming. *Genetic programming and evolvable machines*, 1(1-2):95–119, 2000.

K. Levenberg. A method for the solution of certain non-linear problems in least squares. *Quarterly Journal of Applied Mathmatics*, II(2):164–168, 1944.

Seppo Linnainmaa. Taylor expansion of the accumulated rounding error. *BIT Numerical Mathematics*, 16(2):146–160, 1976.

Sean Luke. Two fast tree-creation algorithms for genetic programming. *IEEE Transactions on Evolutionary Computation*, 4(3):274–283, September 2000a.

Sean Luke. *Issues in scaling genetic programming: breeding strategies, tree generation, and code bloat*. PhD thesis, research directed by Dept. of Computer Science.University of Maryland, College Park, 2000b.

Sean Luke. ECJ: A java-based evolutionary computation research system, 2002. URL http://cs.gmu.edu/~eclab/projects/ecj/.

Sean Luke and Liviu Panait. A survey and comparison of tree generation algorithms. In Lee Spector, Erik D. Goodman, Annie Wu, W. B. Langdon, Hans-Michael Voigt, Mitsuo Gen, Sandip Sen, Marco Dorigo, Shahram Pezeshk, Max H. Garzon, and Edmund Burke, editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, pages 81–88, San Francisco, California, USA, 7-11 July 2001. Morgan Kaufmann.

Sean Luke and Liviu Panait. A comparison of bloat control methods for genetic programming. *Evolutionary Computation*, 14(3):309–344, 2006.

Hammad Majeed and Conor Ryan. A less destructive, context-aware crossover operator for GP. In *Genetic Programming*, volume 3905 of *Lecture Notes in Computer Science*, pages 36–48. Springer Berlin / Heidelberg, 2006.

Donald W Marquardt. An algorithm for least-squares estimation of nonlinear parameters. *Journal of the Society for Industrial & Applied Mathematics*, 11(2):431–441, 1963.

Trent McConaghy. FFX fast, scalable, deterministic symbolic regression technology. In *Genetic Programming Theory and Practice IX*, pages 235–260. Springer, 2011.

Peter McCullagh and John A Nelder. *Generalized linear models*, volume 37. CRC press, 1989.

Robert I Mckay, Nguyen Xuan Hoai, Peter Alexander Whigham, Yin Shan, and Michael O'Neill. Grammar-based genetic programming: a survey. *Genetic Programming and Evolvable Machines*, 11(3-4):365–396, 2010.

Julian F Miller and Peter Thomson. Cartesian genetic programming. In *Genetic Programming*, pages 121–132. Springer, 2000.

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

Pablo Moscato et al. On evolution, search, optimization, genetic algorithms and martial arts: Towards memetic algorithms. *Caltech concurrent computation program, C3P Report*, 826:1989, 1989.

Shreya Mukherjee and Margaret J. Eppstein. Differential evolution of constants in genetic programming improves efficacy and bloat. In Katya Rodriguez and Christian Blum, editors, *GECCO 2012 Late breaking abstracts workshop*, pages 625–626, Philadelphia, Pennsylvania, USA, 7-11 July 2012. ACM.

John A Nelder and Roger Mead. A simplex method for function minimization. *The computer journal*, 7(4):308–313, 1965.

Kourosh Neshatian, Mengjie Zhang, and Peter Andreae. A filter approach to multiple feature construction for symbolic learning classifiers using genetic programming. *IEEE Transactions on Evolutionary Computation*, 16(5): 645–661, 2012.

Christoph Neumüller, Andreas Scheibenpflug, Stefan Wagner, Andreas Beham, and Michael Affenzeller. Large scale parameter meta-optimization of metaheuristic optimization algorithms with heuristiclab hive. *Actas del VIII Español sobre Metaheurísticas, Algoritmos Evolutivos y Bioinspirados (MAEB). Albacete, Spain*, 2012.

Quang Uy Nguyen, Xuan Hoai Nguyen, and Michael O'Neill. Semantic aware crossover for genetic programming: the case for real-valued function regression. In *European Conference on Genetic Programming*, pages 292–302. Springer, 2009.

Michael O'Neill and Conor Ryan. *Grammatical evolution: evolutionary automatic programming in an arbitrary language*, volume 4. Springer Science & Business Media, 2012.

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

José Antonio Parejo, Antonio Ruiz-Cortés, Sebastián Lozano, and Pablo Fernandez. Metaheuristic optimization frameworks: a survey and benchmarking. *Soft Computing*, 16(3):527–561, 2012.

Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *The Journal of Machine Learning Research*, 12:2825–2830, 2011.

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

Riccardo Poli. A simple but theoretically-motivated method to control bloat in genetic programming. In *Proceedings of the 6th European conference on Genetic programming*, EuroGP'03, pages 204–217, Berlin, Heidelberg, 2003. Springer-Verlag.

Riccardo Poli. Covariant tarpeian method for bloat control in genetic programming. *Genetic Programming Theory and Practice VIII*, 8:71–90, 2010.

Riccardo Poli and William B Langdon. Schema theory for genetic programming with one-point crossover and point mutation. *Evolutionary Computation*, 6(3):231–252, 1998a.

Riccardo Poli and William B Langdon. On the search properties of different crossover operators in genetic programming. *Genetic Programming*, pages 293–301, 1998b.

Riccardo Poli, William B. Langdon, and Nicholas Freitag McPhee. *A field guide to genetic programming*. Published via `http://lulu.com` and freely available at `http://www.gp-field-guide.org.uk`, 2008.

Michael JD Powell. An efficient method for finding the minimum of a function of several variables without calculating derivatives. *The computer journal*, 7(2):155–162, 1964.

Markus Quade, Markus Abel, Kamran Shafi, Robert K Niven, and Bernd R Noack. Prediction of dynamical systems by symbolic regression. *Physical Review E*, 94(1):012214, 2016.

Gunther R Raidl. A hybrid GP approach for numerically robust symbolic regression. *Genetic Programming*, pages 323–328, 1998.

Louis B. Rall. *Automatic Differentiation: Techniques and Applications*, volume 120 of *Lecture Notes in Computer Science*. Springer, Berlin, 1981. ISBN 0–540–10861–0.

Carl Edward Rasmussen. Gaussian processes in machine learning. In *Advanced lectures on machine learning*, pages 63–71. Springer, 2004.

Conor Ryan and Maarten Keijzer. An analysis of diversity of constants of genetic programming. In Conor Ryan, Terence Soule, Maarten Keijzer, Edward Tsang, Riccardo Poli, and Ernesto Costa, editors, *Genetic Programming, Proceedings of EuroGP'2003*, volume 2610 of *LNCS*, pages 404–413, Essex, 14-16 April 2003. Springer-Verlag.

Michael Schmidt and Hod Lipson. Distilling free-form natural laws from experimental data. *science*, 324(5923):81–85, 2009.

Michael Schmidt and Hod Lipson. Age-fitness pareto optimization. In *Genetic Programming Theory and Practice VIII*, pages 129–146. Springer, 2011.

Michael D. Schmidt and Hod Lipson. Co-evolving fitness predictors for accelerating and reducing evaluations. In Rick L. Riolo, Terence Soule, and Bill Worzel, editors, *Genetic Programming Theory and Practice IV*, volume 5 of *Genetic and Evolutionary Computation*, pages 113–130. Springer, Ann Arbor, 2006.

Michael D. Schmidt and Hod Lipson. Coevolution of fitness predictors. *IEEE Trans. on Evolutionary Computation*, pages 736–749, 2008.

Marc Schoenauer, Michele Sebag, Francois Jouve, Bertrand Lamy, and Habibou Maitournam. Evolutionary identification of macro-mechanical models. In Peter J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 23, pages 467–488. MIT Press, Cambridge, MA, USA, 1996.

Hans-Paul Schwefel. *Numerical optimization of computer models*. John Wiley & Sons, Inc., 1981.

Dominic Searson. *Non-linear PLS using genetic programming*. PhD thesis, School of Chemical Engineering and Advanced Materials University of Newcastle upon Tyne An online version of a thesis submitted to the Faculty of Engineering, University of Newcastle upon Tyne, 2002.

Dominic P Searson. GPTIPS 2: an open-source software platform for symbolic data mining. In *Handbook of genetic programming applications*, pages 551–573. Springer, 2015.

Dominic P Searson, 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*, volume 1, pages 77–80. Citeseer, 2010.

Ken C. Sharman, Anna I. Esparcia Alcazar, and Yun Li. Evolving signal processing algorithms by genetic programming. In A. M. S. Zalzala, editor, *First International Conference on Genetic Algorithms in Engineering Systems: Innovations and Applications, GALESIA*, volume 414, pages 473–480, Sheffield, UK, 1995. IEE.

Alex Shtof, Alexander Agathos, Yotam Gingold, Ariel Shamir, and Daniel Cohen-Or. Geosemantic Snapping for Sketch-Based Modeling. *Computer Graphics Forum*, 32(2):245–253, 2013.

Sara Silva and Jonas Almeida. Dynamic maximum tree depth. In *Genetic and Evolutionary Computation Conference*, pages 1776–1787. Springer, 2003.

Sara Silva and Ernesto Costa. Dynamic limits for bloat control in genetic programming and a review of past and current bloat theories. *Genetic Programming and Evolvable Machines*, 10(2):141–179, 2009.

Guido F Smits and Mark Kotanchek. Pareto-front exploitation in symbolic regression. In *Genetic programming theory and practice II*, pages 283–299. Springer, 2005.

Nidamarthi Srinivas and Kalyanmoy Deb. Muiltiobjective optimization using nondominated sorting in genetic algorithms. *Evolutionary computation*, 2 (3):221–248, 1994.

Sean Stijven, Wouter Minnebo, and Katya Vladislavleva. Separating the wheat from the chaff: on feature selection and feature importance in regression random forests and symbolic regression. In Steven Gustafson and Ekaterina Vladislavleva, editors, *Proceedings of the 13th annual conference companion on Genetic and evolutionary computation*, pages 623–630, Dublin, Ireland, 2011. ACM.

Rainer Storn and Kenneth Price. Differential evolution–a simple and efficient heuristic for global optimization over continuous spaces. *Journal of global optimization*, 11(4):341–359, 1997.

Astro Teller. The evolution of mental models. In *Advances in Genetic Programming*, pages 199–217. MIT Press, 1994.

Robert Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society. Series B (Methodological)*, pages 267–288, 1996.

Alexander Topchy and William F. Punch. Faster genetic programming based on local gradient search of numeric leaf values. In Lee Spector, Erik D. Goodman, Annie Wu, W. B. Langdon, Hans-Michael Voigt, Mitsuo Gen, Sandip Sen, Marco Dorigo, Shahram Pezeshk, Max H. Garzon, and Edmund Burke, editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, pages 155–162, San Francisco, California, USA, 2001. Morgan Kaufmann.

Vassili V. Toropov and Luis F. Alvarez. Application of genetic programming to the choice of a structure of multipoint approximations. In *1st ISSMO/NASA Internet Conf. on Approximations and Fast Reanalysis in Engineering Optimization*, 1998.

Nguyen Quang Uy, Michael O'Neill, Nguyen Xuan Hoai, Bob Mckay, and Edgar Galván-López. Semantic similarity based crossover in GP: the case for real-valued function regression. In *Proceedings of the 9^{th} international conference on Artificial evolution*, EA'09, pages 170–181, Berlin, Heidelberg, 2010. Springer-Verlag.

Stefan Van Der Walt, S Chris Colbert, and Gael Varoquaux. The NumPy array: a structure for efficient numerical computation. *Computing in Science & Engineering*, 13(2):22–30, 2011.

Leonardo Vanneschi, Mauro Castelli, and Sara Silva. Measuring bloat, overfitting and functional complexity in genetic programming. In *Proceedings of the 12^{th} annual conference on Genetic and evolutionary computation*, pages 877–884. ACM, 2010.

Vladimir Vapnik. *The Nature of Statistical Learning Theory*. Springer Science & Business Media, 1999.

Ekaterina Vladislavleva, Guido Smits, and Dick den Hertog. On the importance of data balancing for symbolic regression. *IEEE Transactions on Evolutionary Computation*, 14(2):252–277, 2010. ISSN 1089-778X.

Ekaterina J Vladislavleva, Guido F Smits, and Dick Den Hertog. Order of nonlinearity as a complexity measure for models generated by symbolic

regression via pareto genetic programming. *Evolutionary Computation, IEEE Transactions on*, 13(2):333–349, 2009.

Stefan Wagner. *Heuristic Optimization Software Systems - Modeling of Heuristic Optimization Algorithms in the HeuristicLab Software Environment*. PhD thesis, Institute for Formal Models and Verification, Johannes Kepler University, Linz, Austria, 2009.

Stefan Wagner and Michael Affenzeller. Sexualga: Gender-specific selection for genetic algorithms. In *Proceedings of the 9th World Multi-Conference on Systemics, Cybernetics and Informatics (WMSCI)*, volume 4, pages 76–81, 2005.

Stefan Wagner, Gabriel Kronberger, Andreas Beham, Michael Kommenda, Andreas Scheibenpflug, Erik Pitzer, Stefan Vonolfen, Monika Kofler, S Winkler, Viktoria Dorfer, et al. Architecture and design of the heuristiclab optimization environment. In *Advanced Methods and Applications in Computational Intelligence*, pages 197–261. Springer, 2014.

Pu Wang, Ke Tang, Edward PK Tsang, and Xin Yao. A memetic genetic programming with decision tree-based local search for classification problems. In *Evolutionary Computation (CEC), 2011 IEEE Congress on*, pages 917–924. IEEE, 2011.

David R White. Software review: the ECJ toolkit. *Genetic Programming and Evolvable Machines*, 13(1):65–67, 2012.

David R. White, James McDermott, Mauro Castelli, Luca Manzoni, Brian W. Goldman, Gabriel Kronberger, Wojciech Jaskowski, Una-May O'Reilly, and Sean Luke. Better GP benchmarks: community survey results and proposals. *Genetic Programming and Evolvable Machines*, 14(1): 3–29, March 2013. ISSN 1389-2576. doi: doi:10.1007/s10710-012-9177-2.

Stephan M. Winkler. *Evolutionary System Identification - Modern Concepts and Practical Applications*. PhD thesis, Institute for Formal Models and Verification, Johannes Kepler University, Linz, Austria, April 2008.

Tony Worm and Kenneth Chiu. Prioritized grammar enumeration: symbolic regression by dynamic programming. In *Proceedings of the 15th annual conference on Genetic and evolutionary computation*, pages 1021–1028. ACM, 2013.

Huayang Xie. *An Analysis of Selection in Genetic Programming*. PhD thesis, Victoria University of Wellington, 2009.

Emigdio Z-Flores, Leonardo Trujillo, Oliver Schütze, Pierrick Legrand, et al. Evaluating the effects of local search in genetic programming. In *EVOLVE-A Bridge between Probability, Set Oriented Numerics, and Evolutionary Computation V*, pages 213–228. Springer, Cham, 2014.

Emigdio Z-Flores, Leonardo Trujillo, Oliver Schütze, Pierrick Legrand, et al. A local search approach to genetic programming for binary classification. In *Proceedings of the 2015 on Genetic and Evolutionary Computation Conference-GECCO'15*, 2015.

Qiongyun Zhang, Chi Zhou, Weimin Xiao, and Peter C. Nelson. Improving gene expression programming performance by using differential evolution. In *Sixth International Conference on Machine Learning and Applications, ICMLA 2007*, pages 31–37, Cincinnati, Ohio, USA, 13-15 December 2007. IEEE. doi: doi:10.1109/ICMLA.2007.62.

Douglas Zongker and Bill Punch. lil-gp 1.1 genetic programming system, 09 1998. URL http://garage.cse.msu.edu/software/lil-gp/index.html.

Hui Zou and Trevor Hastie. Regularization and variable selection via the elastic net. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 67(2):301–320, 2005.

List of Figures

List of Tables

Michael Kommenda

Curriculum vitae

Personal Data

Date of Birth	25.08.1983
Nationality	Austrian
Civil Status	Single

Education

2011–2018 **PhD Studies in Computer Sciences**, *Johannes Kepler University*, Linz, Austria.
Thesis: Local Optimization and Complexity Control in Symbolic Regression

2009–2011 **Master Studies in Bioinformatics**, *University of Applied Sciences Upper Austria*, Hagenberg, Austria.
Thesis: Using Symbolic Regression to model a Blast Furnace and Temper Mill Process

2003–2007 **Diploma Studies in Bioinformatics**, *University of Applied Sciences Upper Austria*, Hagenberg, Austria.
Thesis: Evaluierung alignmentbasierter Duplikaterkennung in integrierten PPI-Datenbeständen

1997–2002 **Higher School Certificate**, *Upper secondary technical and vocational college, Department for informatics and software engineering*, Leonding, Austria.

Experience

2007–present **Research Asssociate**, *University of Applied Sciences Upper Austria*, Hagenberg.
Working in various research projects in the context of data-based modeling and data analaysis. Developer of the open-source framework HeuristicLab.

2011–present **Teaching Assistent**, *University of Applied Sciences Upper Austria*, Hagenberg.
Teaching students version control, issue and project management in software projects.

2004–2005 **Tutor in C++**, *University of Applied Sciences Upper Austria*, Hagenberg, Austria.

Publications

2017 Affenzeller, Michael, Bogdan Burlacu, Stephan Winkler, Michael Kommenda, Gabriel Kronberger, and Stefan Wagner (2017). "Offspring Selection Genetic Algorithm Revisited: Improvements in Efficiency by Early Stopping Criteria in the Evaluation of Unsuccessful Individuals". In: *International Conference on Computer Aided Systems Theory*. Springer, Cham, pp. 424–431.

Affenzeller, Michael, Stephan M Winkler, Bogdan Burlacu, Gabriel Kronberger, Michael Kommenda, and Stefan Wagner (2017). "Dynamic observation of genotypic and phenotypic diversity for different symbolic regression GP variants". In: *Proceedings of the Genetic and Evolutionary Computation Conference Companion*. ACM, pp. 1553–1558.

Burlacu, Bogdan, Michael Affenzeller, Michael Kommenda, Gabriel Kronberger, and Stephan Winkler (2017). "Analysis of Schema Frequencies in Genetic Programming". In: *International Conference on Computer Aided Systems Theory*. Springer, Cham, pp. 432–438.

Karder, Johannes, Stefan Wagner, Andreas Beham, Michael Kommenda, and Michael Affenzeller (2017). "Towards the design and implementation of optimization networks in HeuristicLab". In: *Proceedings of the Genetic and Evolutionary Computation Conference Companion*. ACM, pp. 1209–1214.

Kommenda, Michael, Johannes Karder, Andreas Beham, Bogdan Burlacu, Gabriel Kronberger, Stefan Wagner, and Michael Affenzeller (2017). "Optimization Networks for Integrated Machine Learning". In: *International Conference on Computer Aided Systems Theory*. Springer, Cham, pp. 392–399.

Kronberger, Gabriel, Bogdan Burlacu, Michael Kommenda, Stephan Winkler, and Michael Affenzeller (2017). "Measures for the Evaluation and Comparison of Graphical Model Structures". In: *International Conference on Computer Aided Systems Theory*. Springer, pp. 283–290.

2016 Kommenda, Michael, Gabriel Kronberger, Michael Affenzeller, Stephan M Winkler, and Bogdan Burlacu (2016). "Evolving simple symbolic regression models by multi-objective genetic programming". In: *Genetic Programming Theory and Practice XIII*. Springer, pp. 1–19.

2015 Affenzeller, Michael, Andreas Beham, Stefan Vonolfen, Erik Pitzer, Stephan M Winkler, Stephan Hutterer, Michael Kommenda, Monika Kofler, Gabriel Kronberger, and Stefan Wagner (2015). "Simulation-Based Optimization with HeuristicLab: Practical Guidelines and Real-World Applications". In: *Applied Simulation and Optimization*. Springer, pp. 3–38.

Beham, Andreas, Judith Fechter, Michael Kommenda, Stefan Wagner, Stephan M Winkler, and Michael Affenzeller (2015). "Optimization strategies for integrated knapsack and traveling salesman problems". In: *International Conference on Computer Aided Systems Theory*. Springer, pp. 359–366.

Burlacu, Bogdan, Michael Affenzeller, and Michael Kommenda (2015). "On the Effectiveness of Genetic Operations in Symbolic Regression". In: *International Conference on Computer Aided Systems Theory*. Springer, pp. 367–374.

Burlacu, Bogdan, Michael Affenzeller, Stephan Winkler, Michael Kommenda, and Gabriel Kronberger (2015). "Methods for genealogy and building block analysis in genetic programming". In: *Computational Intelligence and Efficiency in Engineering Systems*. Springer, pp. 61–74.

Burlacu, Bogdan, Michael Kommenda, and Michael Affenzeller (2015). "Building Blocks Identification Based on Subtree Sample Counts for Genetic Programming". In: *Computer Aided System Engineering (APCASE), 2015 Asia-Pacific Conference on*. IEEE, pp. 152–157.

Kommenda, Michael, Michael Affenzeller, Gabriel Kronberger, Bogdan Burlacu, and Stephan Winkler (2015). "Multi-population genetic programming with data migration for symbolic regression". In: *Computational Intelligence and Efficiency in Engineering Systems*. Springer, pp. 75–87.

Kommenda, Michael, Andreas Beham, Michael Affenzeller, and Gabriel Kronberger (2015). "Complexity Measures for Multi-objective Symbolic Regression". In: *International Conference on Computer Aided Systems Theory*. Springer, pp. 409–416.

Kronberger, Gabriel and Michael Kommenda (2015). "Search Strategies for Grammatical Optimization Problems—Alternatives to Grammar-Guided Genetic Programming". In: *Computational Intelligence and Efficiency in Engineering Systems*. Springer, pp. 89–102.

Kronberger, Gabriel, Michael Kommenda, Stephan Winkler, and Michael Affenzeller (2015). "Using Contextual Information in Sequential Search for Grammatical Optimization Problems". In: *International Conference on Computer Aided Systems Theory*. Springer, pp. 417–424.

Scheibenpflug, Andreas, Andreas Beham, Michael Kommenda, Johannes Karder, Stefan Wagner, and Michael Affenzeller (2015). "Simplifying Problem Definitions in the HeuristicLab Optimization Environment". In: *Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation*. ACM, pp. 1101–1108.

Winkler, Stephan M, Michael Affenzeller, Gabriel Kronberger, Michael Kommenda, Bogdan Burlacu, and Stefan Wagner (2015). "Sliding window symbolic regression for detecting changes of system dynamics". In: *Genetic Programming Theory and Practice XII*. Springer, pp. 91–107.

Winkler, Stephan M, Gabriel Kronberger, Michael Kommenda, Stefan Fink, and Michael Affenzeller (2015). "Dynamics of Predictability and Variable Influences Identified in Financial Data Using Sliding Window Machine Learning". In: *International Conference on Computer Aided Systems Theory*. Springer, pp. 326–333.

2014 Affenzeller, Michael, Stephan M Winkler, Gabriel Kronberger, Michael Kommenda, Bogdan Burlacu, and Stefan Wagner (2014). "Gaining deeper insights in symbolic regression". In: *Genetic Programming Theory and Practice XI*. Springer, pp. 175–190.

Beham, Andreas, Johannes Karder, Gabriel Kronberger, Stefan Wagner, Michael Kommenda, and Andreas Scheibenpflug (2014). "Scripting and framework integration in heuristic optimization environments". In: *Proceedings of the Companion Publication of the 2014 Annual Conference on Genetic and Evolutionary Computation*. ACM, pp. 1109–1116.

Bramerdorfer, Gerd, Stephan M Winkler, Michael Kommenda, Guenther Weidenholzer, Siegfried Silber, Gabriel Kronberger, Michael Affenzeller, and Wolfgang Amrhein (2014). "Using FE calculations and data-based system identification techniques to model the nonlinear behavior of PMSMs". In: *IEEE Transactions on Industrial Electronics* 61.11, pp. 6454–6462.

Kommenda, Michael, Michael Affenzeller, Bogdan Burlacu, Gabriel Kronberger, and Stephan M Winkler (2014). "Genetic programming with data migration for symbolic regression". In: *Proceedings of the Companion Publication of the 2014 Annual Conference on Genetic and Evolutionary Computation*. ACM, pp. 1361–1366.

Wagner, Stefan, Gabriel Kronberger, Andreas Beham, Michael Kommenda, Andreas Scheibenpflug, Erik Pitzer, Stefan Vonolfen, Monika Kofler, Stephan Winkler, Viktoria Dorfer, et al. (2014). "Architecture and design of the heuristiclab optimization environment". In: *Advanced Methods and Applications in Computational Intelligence*. Springer, pp. 197–261.

Winkler, Stephan M, Michael Affenzeller, Gabriel K Kronberger, Michael Kommenda, Stefan Wagner, Witold Jacak, and Herbert Stekel (2014). "On the Identification of Virtual Tumor Markers and Tumor Diagnosis Predictors Using Evolutionary Algorithms". In: *Advanced Methods and Applications in Computational Intelligence*. Springer, pp. 95–122.

2013 Burlacu, Bogdan, Michael Affenzeller, and Michael Kommenda (2013). "On the evolutionary behavior of genetic programming with constants optimization". In: *International Conference on Computer Aided Systems Theory*. Springer, pp. 284–291.

Burlacu, Bogdan, Michael Affenzeller, Michael Kommenda, Stephan Winkler, and Gabriel Kronberger (2013). "Visualization of genetic lineages and inheritance information in genetic programming". In: *Proceedings of the 15th annual conference companion on Genetic and evolutionary computation*. ACM, pp. 1351–1358.

Kommenda, Michael, Michael Affenzeller, Gabriel Kronberger, and Stephan M Winkler (2013). "Nonlinear Least Squares Optimization of Constants in Symbolic Regression". In: *International Conference on Computer Aided Systems Theory*. Springer, pp. 420–427.

Kommenda, Michael, Gabriel Kronberger, Stephan Winkler, Michael Affenzeller, and Stefan Wagner (2013). "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*. ACM, pp. 1121–1128.

Kronberger, Gabriel and Michael Kommenda (2013). "Evolution of covariance functions for gaussian process regression using genetic programming". In: *International Conference on Computer Aided Systems Theory*. Springer, pp. 308–315.

Kronberger, Gabriel, Michael Kommenda, Stefan Wagner, and Heinz Dobler (2013). "GPDL: a framework-independent problem definition language for grammar-guided genetic programming". In: *Proceedings of the 15th annual conference companion on Genetic and evolutionary computation*. ACM, pp. 1333–1340.

Rid, Raphaela, Wolfgang Strasser, Doris Siegl, Christian Frech, Michael Kommenda, Thomas Kern, Helmut Hintner, Johann W Bauer, and Kamil Önder (2013). "PRIMOS: An Integrated Database of Reassessed Protein–Protein Interactions Providing Web-Based Access to In Silico Validation of Experimentally Derived Data". In: *Assay and drug development technologies* 11.5, pp. 333–346.

Vonolfen, Stefan, Andreas Beham, Michael Kommenda, and Michael Affenzeller (2013). "Structural synthesis of dispatching rules for dynamic dial-a-ride problems". In: *International Conference on Computer Aided Systems Theory*. Springer, Berlin, Heidelberg, pp. 276–283.

Winkler, Stephan M, Michael Affenzeller, Gabriel Kronberger, Michael Kommenda, Stefan Wagner, Viktoria Dorfer, Witold Jacak, and Herbert Stekel (2013). "On the use of estimated tumour marker classifications in tumour diagnosis prediction–a case study for breast cancer". In: *International Journal of Simulation and Process Modelling* 8.1, pp. 29–41.

2012 Kommenda, Michael, Gabriel Kronberger, Stefan Wagner, Stephan Winkler, and Michael Affenzeller (2012). "On the architecture and implementation of tree-based genetic programming in HeuristicLab". In: *Proceedings of the 14th annual conference companion on Genetic and evolutionary computation*. ACM, pp. 101–108.

Kronberger, Gabriel, Stefan Wagner, Michael Kommenda, Andreas Beham, Andreas Scheibenpflug, and Michael Affenzeller (2012). "Knowledge discovery through symbolic regression with heuristiclab". In: *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer, Berlin, Heidelberg, pp. 824–827.

Winkler, Stephan, Michael Affenzeller, Stefan Wagner, Gabriel Kronberger, and Michael Kommenda (2012). "Using genetic programming in nonlinear model identification". In: *Identification for Automotive Systems*. Springer, pp. 89–109.

2011 Kommenda, Michael, Gabriel Kronberger, Christoph Feilmayr, and Michael Affenzeller (2011). "Data mining using unguided symbolic regression on a blast furnace dataset". In: *European Conference on the Applications of Evolutionary Computation*. Springer, pp. 274–283.

Kommenda, Michael, Gabriel Kronberger, Christoph Feilmayr, Leonhard Schickmair, Michael Affenzeller, Stephan M Winkler, and Stefan Wagner (2011). "Application of symbolic regression on blast furnace and temper mill datasets". In: *International Conference on Computer Aided Systems Theory*. Springer, pp. 400–407.

Kronberger, Gabriel, Stefan Fink, Michael Kommenda, and Michael Affenzeller (2011). "Macro-economic time series modeling and interaction networks". In: *European Conference on the Applications of Evolutionary Computation*. Springer, Berlin, Heidelberg, pp. 101–110.

Kronberger, Gabriel, Michael Kommenda, and Michael Affenzeller (2011). "Overfitting detection and adaptive covariant parsimony pressure for symbolic regression". In: *Proceedings of the 13th annual conference companion on Genetic and evolutionary computation*. ACM, pp. 631–638.

Winkler, Stephan M, Michael Affenzeller, Gabriel Kronberger, Michael Kommenda, Stefan Wagner, Witold Jacak, and Herbert Stekel (2011). "Analysis of selected evolutionary algorithms in feature selection and parameter optimization for data based tumor marker modeling". In: *International Conference on Computer Aided Systems Theory*. Springer, pp. 335–342.

Zăvoianu, Alexandru-Ciprian, Gabriel Kronberger, Michael Kommenda, Daniela Zaharie, and Michael Affenzeller (2011). "Improving the parsimony of regression models for an enhanced genetic programming process". In: *International Conference on Computer Aided Systems Theory*. Springer, pp. 264–271.

2010 Wagner, Stefan, Andreas Beham, Gabriel Kronberger, Michael Kommenda, Erik Pitzer, Monika Kofler, Stefan Vonolfen, Stephan Winkler, Viktoria Dorfer, and Michael Affenzeller (2010). "HeuristicLab 3.3: A unified approach to metaheuristic optimization". In: *Actas del séptimo congreso español sobre Metaheurísticas, Algoritmos Evolutivos y Bioinspirados (MAEB'2010)*, p. 8.

Winkler, S, M Affenzeller, G Kronberger, M Kommenda, S Wagner, W Jacak, and H Stekel (2010). "Feature selection in the analysis of tumor marker data using evolutionary algorithms". In: *Proceedings of the 7th international mediterranean and latin american modelling multiconference*, pp. 1–6.

2009 Frech, Christian, Michael Kommenda, Viktoria Dorfer, Thomas Kern, Helmut Hintner, Johann W Bauer, and Kamil Önder (2009). "Improved homology-driven computational validation of protein-protein interactions motivated by the evolutionary gene duplication and divergence hypothesis". In: *BMC bioinformatics* 10.1, p. 21.

Kommenda, Michael, Gabriel Kronberger, Stephan Winkler, Michael Affenzeller, Stefan Wagner, Leonhard Schickmair, and Benjamin Lindner (2009). "Application of genetic programming on temper mill datasets". In: *Logistics and Industrial Informatics, 2009. LINDI 2009. 2nd International*. IEEE, pp. 1–5.

Kronberger, Gabriel, Christoph Feilmayr, Michael Kommenda, Stephan Winkler, Michael Affenzeller, and Thomas Burgler (2009). "System identification of blast furnace processes with genetic programming". In: *Logistics and Industrial Informatics, 2009. LINDI 2009. 2nd International*. IEEE, pp. 1–6.