UNIVERSIDADE FEDERAL DE MINAS GERAIS
Escola de Engenharia
PPGEE - Programa de Pós-Graduação em Engenharia Elétrica

Áthila Rocha Trindade

A new SMBO-Based Parameter Tuning Framework to Optimization Algorithms

Belo Horizonte
2019

Áthila Rocha Trindade

A new SMBO-Based Parameter Tuning Framework to Optimization Algorithms

Versão Final

Tese apresentada ao Programa de Pós-Graduação em Engenharia Elétrica da Universidade Federal de Minas Gerais, como requisito parcial à obtenção do título de Doutor em Engenharia Elétrica; Área de Concentração em Sistemas de Computação e Telecomunicações; Linha de Pesquisa: Otimização.

Orientador: Prof. Dr. Felipe Campelo França Pinto

Belo Horizonte
2019

Trindade, Áthila Rocha.
T833n A new SMBO-Based parameter tuning framework to optimization algorithms [recurso eletrônico] / Áthila Rocha Trindade. - 2019.
1 recurso online (vi, 80 f. : il., color.) : pdf.

Orientador: Felipe Campelo França Pinto.

Tese (doutorado) - Universidade Federal de Minas Gerais, Escola de Engenharia.

Apêndices: f.66-71.
Bibliografia: f.72-80.

Exigências do sistema: Adobe Acrobat Reader.

1. Engenharia Elétrica - Teses. 2. Otimização combinatória - Teses. 3. Programação heurística – Teses. 4. R (Linguagem de computador) – Teses. I. Pinto, Felipe Campelo França. II. Universidade Federal de Minas Gerais. Escola de Engenharia. III.Título.

CDU: 621.3(043)

Ficha catalográfica: Biblioteca Profº Mário Werneck, Escola de Engenharia da UFMG

"A New Smbo-based Parameter Tuning Framework to Optimization Algorithms"

Athila Rocha Trindade

Tese de Doutorado submetida à Banca Examinadora designada pelo Colegiado do Programa de Pós-Graduação em Engenharia Elétrica da Escola de Engenharia da Universidade Federal de Minas Gerais, como requisito para obtenção do grau de Doutor em Engenharia Elétrica.

Aprovada em 20 de novembro de 2019.

Por:

Prof. Dr. Felipe Campelo França Pinto
DEE (UFMG) - Orientador

Prof. Dr. Lucas de Souza Batista
DEE (UFMG)

Prof. Dr. Cristiano Leite de Castro
DEE (UFMG)

Prof. Dr. Eduardo Gontijo Carrano
DEE (UFMG)

Prof. Dr. Elizabeth F. Wanner
Computer Science Group (Aston University)

Prof. Dr. Rodrigo Tomás N. Cardoso
Matemática (CEFET-MG)

FEDERAL UNIVERISITY OF MINAS GERAIS

Abstract

A variety of algorithms have been proposed by optimization researchers, for solving several different problems. *Heuristics* and *Metaheuristics* are two class of algorithms which have been widely used for practical optimization problems, due to their capability to achieve good solutions in a feasible runtime when solving a problem, even in problems with high computational complexity (in terms e.g., of modality, non-differentiability, or NP-hardness). These algorithms can have their level of balance of global search and local improvement in the search space tuned by choosing suitable values of a number of user-defined parameters, which can be numerical or categorical.

To take full advantage of the potential of Heuristics and Metaheuristics when solving hard optimization problems, it is necessary to think about appropriate strategies for choose adequate parameter values, that is, parameters values which provide a balance between global search and local improvement in the search space. Former strategies based on choosing parameters values by trial and error did not achieve satisfactory results. In contrast, methods based on statistical modeling and inference regarding algorithm performance, i.e., which recommend parameter values based on generalizations of given statistics observed in samples of problem instances to whole problem classes, have arisen in the past two decades. Algorithm configurations yielded by these methods not only result in better performance for the algorithms, but can also help researchers and practitioners to experimentally investigate certain aspects of algorithmic behavior.

Based on this observation, this work investigates a parameter-tuning framework based on iteratively improving statistical modeling about the algorithm peformance, which represent the algorithm behavior prediction model. The experimental results showed that the current framework is capable to return competitive results of best parameter values when compared to those of some classical parameter tuning methods of the literature, with the advantage of providing prediction models which can reveal the algorithm parameters relevance. This framework is available for using by researchers in the field of general optimization and metaheuristics. It was implemented using the R Language, and is available for using by the scientific community in a form of an open source R package.

UNIVERSIDADE FEDERAL DE MINAS GERAIS

Resumo

Uma variedade de algoritmos tem sido propostos pela área de otimização para resolução dos mais diferentes problemas. *Heurísticas* e *Metaheurísticas* são duas classes de algoritmos que tem sido largamente utilizados para a resolução de problemas práticos de otimização, devido à sua capacidade de alcançar soluções satisfatórias em um tempo computacional razoável, mesmo tratando-se de problemas com alta complexidade computacional (com relação à, por exemplo: problemas não modais, não diferenciáveis ou NP-completos). Estes algoritmos podem ter o balanço entre o nível de exploração global e busca local no espaço de variáveis ajustado através da escolha de adequados valores de parâmetros (os quais podem ser numéricos ou categóricos) que controlam seu comportamento.

Desta forma, para que Heurísticas e Metaheurísticas sejam utilizadas com seu máximo potencial, é necessária a utilização de estratégias apropriadas de escolha dos valores de seus parâmetros, ou seja, devem ser escolhidos valores de parâmetros que balanceiem os níveis de buscas global e local. Estratégias pioneiras baseadas na escolha de parâmetros por tentativa e erro não alcançaram resultados satisfatórios. Por outro lado, métodos baseados em modelagem e inferência estatística sobre os dados de desempenho do algoritmo; isto é, que determinam valores de parâmetros para um algoritmo resolver determinada classe de problemas baseando-se em estatísticas obtidas à partir das amostras de problemas desta classe, tem sido propostos nas últimas duas décadas. Ademais, as configurações (valores de parâmetros) dos algoritmos obtidas por estes métodos podem também auxiliar pesquisadores da área em experimentalmente investigar aspectos do comportamento do algoritmo.

Com base na questão descrita anteriormente, este trabalho investiga a proposição de um novo *framework* de ajuste de parâmetros baseado na melhoria iterativa de modelos estatísticos sobre o desempenho do algoritmo, os quais representam modelos de predição do comportamento do algoritmo. Os resultados experimentais mostraram que o framework é capaz e retornar resultados similares à outros métodos de ajuste de parâmetros difundidos na literatura; com a vantagem de informar também ao usuário modelos de predição do comportamento do algoritmo que descrevem as diferenças de relevância entre os parâmetros. Este framework está disponível para uso dos pesquisadores da área de otimização e metaheurísticas, na forma de um pacote de código aberto da linguagem R.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Since its beginning, humanity has looked for solving all sort of problems in an optimum way. Facing a simple or a complex problem, regardless of its nature, we are interested in minimizing resource use (for instance, minimizing the amount of waiting time when choosing a queue in a supermarket) or maximizing some benefit (for instance, based on an historical analysis, maximizing the financial gain when deciding which stocks should be purchased).

Since the middle of 20th century, researchers belonging to several fields have studied computational methods (or algorithms) for solving several theoretical and practical problems. From the research in computational theory [1], one important mathematical characteristic of these problems came to light: for some of them it is possible to elaborate algorithms that obtain the best solution, irrespective the dimension of the problem, needing an amount of time T that is a polynomial function of the dimension N of the problem. These problems are classified in the literature of Computer Science as having a *polynomial-time complexity* ([1]) or simply as *Polynomial Problems*. Strictly, polynomial problems are such that can be solved by a deterministic Turing Machine in polynomial time.

On the other hand, for others, it seems that it is not possible to elaborate algorithms to obtain the best solution in polynomial time. These problems are classified as having a *Non-Polynomial-time complexity* ([1]) or simply as *Non-Polynomial Problems* or *NP-problems*. Strictly, non-polynomial problems are such that can be solved in a polynomial time only by using a theoretical non-deterministic Turing Machine. In light of this, the task of searching for the best solution of a problem by algorithms has a threshold time related to the mathematical nature of the problem. For Polynomial Problems it is rather reasonable to elaborate algorithms to obtain the optimal solutions, whereas for NP-problems the time required to obtain the optimal solution is often unfeasible.

Consequently, the solution of these problems must often rely on methods that exchange the guarantees of optimality for a lower computational cost to achieve some solution which cannot be proven to be optimal.

Considering the mathematical nature of problems previously mentioned, different types of strategies or algorithms have been used for solving them, according to their computational complexity. Based on Birattari [2], the algorithms can be classified as: *Exact methods*, *Heuristics* and *Metaheuristics*. *Exact methods* have as their main features: **a)** the fact that for a given instance of a problem, for every time they are executed on this instance, they obtain the same solution (the optimum), having a deterministic behavior; and **b)** their specifical character, that is, they are specifically elaborated for solving a certain problem. Their weakness are: **a)** since many problems are Non-Polynomial, the computational time required is unfeasible in practice; and **b)** their specific character. Exact methods are therefore suitable to deal with a large number of Polynomial Problems and with small instances of Non-Polynomial Problems.

Heuristics are algorithms that have as their main characteristics: **a)** they are specifically elaborated for solving a certain problem, not necessarily reaching the best solution. Usually they have a capability to obtain the best or near-best solutions for a large number of instances of a problem, in a feasible computational time; and **b)** They are designed to a specific problem and relies on the specificities of the problem search space. The idea underlying heuristics is to reach a tradeoff between solution quality and computational cost. In this way, they can be used for a considerable number of Non-Polynomial Problems, reaching a satisfactory solution in a feasible runtime. Their weakness is their particular character, that is, an heuristic is implemented to solve a specific problem, as well as the absence of guarantees of optimality.

Metaheuristics are algorithms designed to be used for a variety of problems, rather than for a specific problem. Unlike *exact methods* and *heuristics*, their main characteristics are: **a)** they have a non-deterministic behavior, that is, each run of a given metaheuristic on an instance of a given problem may result in a different solution, and this behavior is thanks to random steps executed while they are searching for the solution; and **b)** they can usually be applied in a reasonable computational time, for a variety of problems. The general character of *metaheuristics* regarding their use is possible due to an implementation based on a philosophy of components: *metaheuristics* can be seen as computational frameworks formed by several components, each one having a specific task concerning the search for solving a problem. These components can be adjusted and combined in different ways, depending on the problem to be solved. Their weakness is that they can have their performance damaged, if their components are poorly adjusted and combined. One of the most broadly used types of metaheuristics are the Evolutionary Algorithms

(EAs), which have been employed for solving optimization problems in several areas of science and engineering ([3], [4], [5], [6],[7]). Non-evolutionary metaheuristics include a wide number of different approaches, like Tabu Search, Simulated Annealing, Ant Colony Optimization, Iterated Local Search (ILS), Variable Neighborhood Search (VNS), etc [2].

Regardless of the type of algorithm being proposed for finding the solution of a problem, its performance has to be measured in some way. Although for all algorithms a theoretical approach can be done in principle, calculating analytically its computational complexity, experimental approaches are generally more effective to evaluate the performance of an algorithm in a practical fashion. Hooker ([8], [9]) argues that a practical analysis brings out how is the expected behavior of an algorithm based on the typical problems that it must solve, in opposition to the theoretical approach, which is based on worst and average-cases, whose frequency of occurrence is often not known. Thinking about measuring the performance of heuristics and metaheuristics, an approach based on *empirical science* suggested by Hooker [8] passes mandatorily by the adoption of techniques of experimental design and statistical analysis, in view of the stochastic character of them.

The performance of metaheuristics is usually controlled by the proper combination of their components, also called *parameters*. When using a metaheuristic to solve a problem, each parameter of it can be adjusted. A bad choice of parameter values can lead the metaheuristic to present an unsatisfactory performance, even if its implementation was done properly. Usually the parameters of a metaheuristic can assume numeric (integer or real) or categorical values from discrete or continuous sets, existing therefore a large number (sometimes an infinite number) of possible combinations of parameter values (each specific combination of parameter values is usually called a *candidate configuration*, or simply a *configuration* of the metaheuristic). Thus, for using a metaheuristic, an optimization problem comes before, which can be generally defined as: *among all the possible candidate configurations, which one leads the metaheuristic to its best performance for a given class of problems?* To answer this question, it is necessary to think about methods for evaluating and comparing several candidate configurations in order to choose the best one.

With the objective of searching for the best candidate configuration of a metaheuristic when solving a problem, two scenarios have been considered in the literature. In the first one, the optimizer aims to predict what is the best (or what are the best) candidate configuration of a metaheuristic for a class of problems, based on the metaheuristic performance on a training set of instances, sampled from the considered class of problems. This task is called in the literature *parameter tuning*. In order to carry out

the parameter tuning, statistical analysis about the performance of different candidate configurations when running them on a training set of instances are obtained, and the best candidate configuration is determined as being that (or those) which presents the best performance. Here, the measure used can varies (for instance, average performance, median performance, etc). In this way, the best candidate configuration would represent the best parameter values for the metaheuristic, when using it for solving instances originated from the considered class of problems.

In the second scenario, it is desirable to determine the best dynamic variation of parameter values of a metaheuristic when running it on a specific instance of a specific problem. This task is called in the literature *parameter control*. In this case, the best parameter values must be chosen in a real time, when the metaheuristic is performing the search itself on an instance of a problem. Therefore, it does not make sense to think about a static set of best parameter values, since they have to vary along the execution of the metaheuristic, according to the search process changes between exploration and exploitation phases. Karafotias, Hoogendoorn and Eiben [10] presented a broad survey about parameter control in Evolutionary Algorithms (EAs). In their work, they argue that parameter tuning and parameter control are related, being the former the stationary side, and the last the nonstationary side of the same coin. Moreover, according to them, there are very good parameter tuning methods developed, publicized and adopted over the last decade, whereas for parameter control the problem is far from being solved.

Specifically for controlling EA parameters, a considerable number of promising parameter control methods have been published, especially those related to self-adaptation techniques in evolution strategies. About their usefullness, according to Birattari [2] the necessity of *parameter control* methods arises when not a training set of instances is available or feasible, and they are suitable when one algorithm is supposed to solve one single instance, typically large and complex. Karafotias, Hoogendoorn and Eiben [10] pointed out that although there is no enough quantitative and theoretical evidence about it, a possibly useful angle to answer this is to consider the application scenario. In their opinion, it could be argued that for repetitive problems parameter tuning is worth the extra effort, whereas parameter control is the logical choice for one-off problems, applications with dynamic fitness functions, and online adaptation, e.g., robotics. Regardless of the difficulties for their development, some examples of parameter control methods for metaheuristics are available in [11], [12], [13], [14], [15], [16], [17], [18], [19], [20], [21], [22], [23], [24] and [25].

Despite the importance of using parameter tuning and control methods, even nowadays is rather relevant the number of works in the literature wherein the metaheuristics calibration is based on poorly justified conventions and ad hoc methods based on some rule

of thumbs ([26]). This phenomenon is particularly easy to observe in the applications literature, but its effects are probably more detrimental for comparison of algorithms, since that bad parameter values may lead to incorrect conclusions about the relative performance of the algorithms. Trying to correct the problems mentioned above, specifically in the field of parameter tuning methods, a number of significant contributions have appeared in the literature over the last two decades ([26],[27], [28], [29], [30], [31], [32], [33], [34], [35], [36]). Among them, those which propose parameter tuning in an automatic way ([26], [28], [31], [32], [33], [34], [35], [36]) should be highlighted. These methods basically involve the evaluation of candidate configurations on a group of instances (training instances), followed by statistical modeling and inference approaches that measure the relative performance of different configurations (usually in terms of mean and median values). The best configurations are chosen to be used by the algorithm for future runs.

Some of the most well-known procedures for tuning metaheuristics are based on racing methods ([2]), such as F-RACE ([34]) and Iterated F-Race (I-F/Race) ([37]), both available as the R-Packages *race* ([33]) and *irace* ([38]). It is also worth highlighting the SMBO (Sequential Models Based Optimization) methods, like BONESA ([39]), SPO ([40]) and SMAC ([41]); and the HyperHeuristic, namely: REVAC ([26]), ParamILS ([35]) and CRS-Tuning ([42]). More details about these methods are also given in the next chapter.

One important aspect of the parameter tuning methods mentioned above is that the majority of them make use of a statistical background that allows them to give consistent predictions about the differences concerning the performances of the candidate configurations. In this manner, failures in designing the statistical experiment may lead to incorrect results by the statistical tests and, consequently, induce the method to give wrong results about the comparison of candidate configurations. Similar configurations may be mistakenly considered distinct, or the results of a statistical comparison may even return an inverted conclusion, declaring an inferior configuration to be significantly superior to another. Furthermore, if the assumptions of the statistical method about the performance data distribution are broken, its accuracy of predicting the performance of configurations possibly will decrease, and this fact can damage the search process for the best configurations. For these reasons, it is important to be aware of better statistical practices and common statistical modelling mistakes to be avoided in the development and improvement of parameter tuning and control techniques.

Specifically about the SMBO methods, one interesting aspect of them is that they can obtain the information about what is the best configuration to be used in a class of problems and a model about the algorithm behavior, describing a relation between the

variation of parameter values and performance values. This model can help the user to understand how the several algorithm parameters influence in the algorithm performance, allowing him/her to identify the most relevant parameters. This work is mainly focused on the aspects of the application of statistical methods to the development and study of tuning metaheuristics strategies based on the SMBO approach. More specifically, this proposal consists in an implementation of a tuning framework with two basic components:

1. A component that builds regression models, based on performance data of candidate configurations, which represent possible algorithm behaviors. It is worth noticing that this component allows the use of several types of regression modelling, including robust methods (those whose accuracy does not have assumptions about the data distributions) and methods that promote the selection of variables, highlighting the most relevant parameters.

2. A component of optimization, which optimizes the regression models built in order to find new promising configurations. Similarly to the prior component, it is possible to use different optimization strategies, like exact methods or even metaheuristics, depending on the nature of the models generated.

The iterative use of these components is the basis of a new tuning framework that has as your main contributions the capability to obtain good candidate configurations joined to a model which represents the algorithm behavior and describes the relative relevance of parameters, regardless of the performance data distribution. As will be discussed more detailed in the next two chapters, several parameter tuning methods neither offer to the user a model representing the algorithm behavior nor the possibility of generating models that reveals the relative relevance of parameters. More formally, the objectives of this work are the following.

1.1 Objectives

1.1.1 General Objectives

1. To investigate the capabilities and weakness of the approaches for tuning metaheuristics.

2. To propose a framework of a new tuning method, based on statistical modeling techniques and inspired on the current state-of-the-art.

1.1.1.1 Specific Objectives

1. To implement a new parameter tuning method, and performs a comparison among its performance and those of the most known parameter tuning methods.

2. To explore experimentally the promising aspects of the new tuning method, named: the use of robust regression modelling and selection of variables; contributing for the scientific evolution concerning parameter tuning to optimization algorithms.

1.2 The Structure of this Work

The remainder of this work is organized as follows: the next chapter details the problem of parameter tuning and presents a description of the most widely used techniques for parameter tuning in the literature. Chapter 3 presents the proposed parameter tuning framework. Chapter 4 presents and discusses the results obtained by the proposed tuning method and the chapter 5 presents the conclusions related to the research and possible future works.

Chapter 2

Parameter Tuning

2.1 Introduction

Considering the tuning process as a tool for supporting the user in choosing the best configuration for a metaheuristic, it is important to keep in mind the potential disadvantages of an *ad hoc* tuning processes. A trial and error based generation of candidate configurations can be rather expensive and inneficient. Moreover, the experience or knowledge of the experimenter about the metaheuristic directly affects his/her capability for generating promising candidate configurations, and can often end up biasing the choice of parameter values according to the user experiences in contexts that are different than those for which the tuning is being performed. This question can be especially problematic when the experimenter has to compare alternatives of different metaheuristics and he/she has a greater expertise about one of them, involuntarily biasing the tuning process.

To overcome these human-related problems, other aspects concerning the tuning process must be elucidated to provide support for its implementation. In a certain way, parameter tuning methods try to "predict the future" (the best configuration for future instances) based on an past experience (evaluations of configurations on a training set), assuming that instances from a training set are *representative*, i.e., that the instances to be sampled in the future are likely to have similar structure to those sampled from the training set. Although there is no mathematical formulation for guaranteeing it, there is a reason for accepting this representativeness as being likely.

Assuming that parameter tuning has its applicability for repetitive problems (instances) belonging to the same class, it is reasonable to expect that the occurrence of instances follows the same rule over the time, such that the characteristics of the instances observed in the present will be likely similar to those observed in the future. Thus, using

parameter tuning as a learning tool, it is possible to learn about the future behavior of a metaheuristic (in relation to a class of problems) based on a training set of instances. Even though the actual rule of generating both the training set and future instances is usually unknown, adopting the same independently and identically distributed sampling for all instances ensures that the empirical mean of the performance on a training set of instances is an unbiased estimator of the expected performance of a metaheuristic on any other future set of instances related to that same class of problems. Consequently, this empirical mean can be used for deriving predictions about the algorithm behavior. Thus, as pointed out by Birattari [2], the assumption of regularity (in terms of their characteristics) concerning the occurrence of instances belonging to the same class of problem is considered here to become more likely the representativeness of a training set of instances. This property is often easy to guarantee in benchmark-based comparisons, since the full population of problem instances is available and the parameters can be fine-tuned for the entire benchmarck set; but more challenging for applied and black-box model based optimization problems.

Another assumption upon which parameter tuning methods rely is that of *independence of observations*. While many researches consider this assumption as stating that metaheuristics executions must be independent, e.g., by setting different initial conditions for the algorithms, this is an incomplete reading of this requirement. In fact, the assumption of independence comes from the statistical modeling that underlies these methods, and requires the absence of significant dependence structures in the residuals of the statistical model being deployed. Violations of this assumption tend to generate effective significance levels that may be very different from those specified for a given statistical procedure, and may arise from a variety of sources - e.g., failure to consider block effects in the modeling, or the incorrect interpretation of what constitutes a replicate in the experimental setting.

Two other important aspects concern the measure and the metric adopted by the method. First, an appropriate cost function has to be defined as a measure. For instance, for the TSP (Travelling Salesman Problem) the measure could be the length of the tour [2], or the time needed for completing the tour could be more appropriate if the traffic conditions are considered. The statistical metric to be applied can also vary, e. g., the third quantile or the median are known to be more robust statistics than the expected value and in some application this property could be particularly appealing [2]. In the case of metaheuristics, as they are stochastic algorithms, a metric of interest is its expected value, which has an unbiased and consistent estimator (the sample mean).

Since parameter tuning methods are proposed for solving a kind of optimization problem (called here of *algorithm configuration problem*), it is necessary to describe it formally,

as a basis from which will be presented the formal notation of the parameter tuning methods. The next section presents this description.

2.2 The Algorithm Configuration Problem

The algorithm configuration problem defined below is based on Birattari [2]. Let the following aspects related to the problem be defined:

- Θ is the set of all possible parameter configurations for a given metaheuristic.
- Γ is the set of all problem instances belonging to a certain problem class.
- $X_{(\theta;\gamma)}$ is a random variable representing the performance of a candidate configuration $\theta \in \Theta$ on a given instance $\gamma \in \Gamma$ (according to a given performance indicator that must be maximized).
- $\mu_{(\theta;\gamma)}$ is the expected value of the performance of a candidate configuration $\theta \in \Theta$ on a given instance $\gamma \in \Gamma$, that is, the expected value of the random variable $X_{(\theta;\gamma)}$.
- $\chi_{(\theta;\Gamma)}$ is the set of values $\mu_{(\theta;\gamma)}$ for all $\gamma \in \Gamma$, that is, the set of expected performances of a candidate configuration θ on all instances belonging to a problem class Γ.
- $\mu_{(\theta;\Gamma)}$ is the expected value of $\chi_{(\theta;\Gamma)}$.

It is then possible to state the *algorithm configuration problem* as:

$$\text{Find } \theta^* = \arg\max_{\theta} \mu_{(\theta;\Gamma)} \tag{2.1}$$

that is, the problem of finding the configuration that maximizes the expected performance of a given metaheuristic for a given problem class.

2.3 Parameter Tuning Methods

Based on the formulation above, the parameter tuning methods try to estimate θ^* by using information obtained from a finite number of problem instances $\gamma_1, ..., \gamma_q$ to predict features of the instance set Γ. In this section, some parameter tuning methods are reviewed.

A variety of different tuning methods have been proposed over the years to determine the best configurations of algorithms when solving a given problem class. Based on their working mechanisms and design principles, it is possible to group these methods in three major categories: racing methods, SMBO methods, and hyper-heuristics. This section describes the most widely used methods from each category.

2.3.1 Racing Methods

Racing methods initially appeared in the Supervised Learning literature, proposed for solving the *model selection problem*. The idea of these methods, introduced by Maron & Moore (Maron & Moore *apud* Birattari [2]), and implemented in their algorithm called *Hoeffding race*, is what characterizes the whole class of racing algorithms: the search for the best model structure can be speeded up by discarding inferior candidates as soon as sufficient evidence is gathered against them.

As the computation goes on, the estimate of the leave-one-out measure for the candidates models gets more accurate and a statistical test of hypothesis can be adopted for deciding whether the observed differences in the leave-one-out estimates of the candidate subsets is significant. If this is the case, the worst candidate models are discarded and the race goes on with the remaining. This implies in a more suitable use of the computational budget, since the best candidate subsets are evaluated more intensively. The *Hoeffding race* algorithm adopted a statistical test based on Hoeffiding's formula concerning the confidence on the empirical mean of k positive numbers c_1,...,c_k, sampled independently from the same distribution, when an upper bound on the random c is known.

An evolution of *Hoeffding race* for the model selection problem is the algorithm *BRACE*, proposed by Moore & Lee (More & Lee *apud* Birattari [2]). BRACE is based on Bayesians statistics and implements a blocking statistical technique to compare the subset candidates.

Based on the ideas presented in Hoeffding and BRACE algorithms, that is, evaluation of candidates based on a racing and using blocking in statistical tests to compare them, the F-Race and I-F/Race algorithms were proposed for the algorithm configuration problem. The next section describes these algorithms.

2.3.1.1 The F-Race and I-F/Race Methods

To address the algorithm configuration problem, Birattari et al. [34] proposed a method called *F-Race*, based on machine learning racing algorithms. The main concept behind this procedure is to evaluate iteratively a given set of candidate configurations on a finite

number of instances, gradually building statistical evidence until concluding that one or more candidate configurations are significantly inferior. Once this is determined, those configurations with worst performance are eliminated and the race goes on only with the surviving ones. The procedure stops when some termination criteria is reached, e.g., a maximum computational budget or a minimum configurations survival.

More specifically, the method evaluates a finite set of candidate configurations step by step. Let $\theta_i \in \Theta$, $i = 1, \ldots, m$ be the candidate configurations and $\gamma_j \in \Gamma$, $j = 1, \ldots, q$ be the instances available for tuning. In the first step of F-Race, each configuration is evaluated on instance γ_1, obtaining a block $b_1 = \left[x_{(\theta_1;\gamma_1)}, x_{(\theta_2;\gamma_1)}, \ldots, x_{(\theta_m;\gamma_1)}\right]$, where $x_{(\theta_i;\gamma_j)}$ represents an observation drawn from $X_{(\theta_i;\gamma_j)}$. After $K_{\min}$ steps, the method has generated $K_{\min}$ blocks:

$$\begin{aligned} b_1 &= \left[x_{(\theta_1;\gamma_1)}, x_{(\theta_2;\gamma_1)}, \ldots, x_{(\theta_m;\gamma_1)}\right] \\ b_2 &= \left[x_{(\theta_1;\gamma_2)}, x_{(\theta_2;\gamma_2)}, \ldots, x_{(\theta_m;\gamma_2)}\right] \\ &\vdots \\ b_{K_{min}} &= \left[x_{(\theta_1;\gamma_{K_{\min}})}, x_{(\theta_2;\gamma_{K_{\min}})}, \ldots, x_{(\theta_m;\gamma_{K_{\min}})}\right] \end{aligned}$$

At this point, the Friedman test [43] is applied to this blocked data. In the Friedman test, within each block b_k, the costs $(x_{(\theta_1;\gamma_k)}, x_{(\theta_2,\gamma_k)}, ...x_{(\theta_m;\gamma_k)})$ are ranked in non-decreasing order. Let R_{θ_i} the sum of ranks for a candidate configuration θ_i, considering the K_{min} instances. The value of the Friedman statistic is calculated as:

$$Fr = \frac{12 \sum_{i=1}^{m} (R_{\theta_i} - \frac{K_{min}(m+1)}{2})^2}{K_{min} m(m+1) - \frac{T}{m-1}} \tag{2.2}$$

where

$$T = \sum_{i=1}^{K_{min}} \sum_{v \in V_i} (nc_v)^3 - (nc_v) \tag{2.3}$$

nc_v = number of ocurrences of observations with an specific value v, on an instance i, considering m candidate configurations.
V_i = set of different values v for the observations that occur for all m candidate configurations on an instance i.

If there are no ties on an instance k, $nc_v = 1 \; \forall \; v \in V_k$.

The term T in the formulation 2.2 is used to eliminate ties cases, if they exist. Under the null hypothesis that all possible rankings of the candidate configurations within each block are equally likely, Fr is approximately χ^2 distributed with $(m-1)$ degrees of freedom. If the observed exceeds the 1-α quantile of the distribution, the null hypothesis is rejected, at the significance level of α, in favor of the hypothesis that at least one candidate tends to yield a better performance than at least one other.

The Friedman test is based on two fundamental assumptions [43, 44]: (i) the blocks have been randomly sample from the population they represent, assuming *unbiasedness* and *independence* between blocks; and (ii) the response variable represents a continuous random variable. While the second one can be frequently guaranteed when tuning metaheuristics, the first one is frequently ignored and a source of pseudoreplication ([45], [46], [47]) in implementations of the racing procedures. This assumption is common to all paired/blocked tests employed in experimental comparisons of metaheuristics, which makes its violations not only a problem for tuning methods, but a wider methodological issue.

Since the Friedman test uses the tuning instances γ_j as blocks, it is important to ensure that the instances used are an *independent* sample of Γ, so that the test can maintain its nominal power and confidence levels. It means that the values μ_{θ_i,γ_j} (expected performance value of a candidate on an instance) sampled from $\chi_{(\theta_i;\Gamma)}$ must be independent. In terms of instances, it follows that each of the q instances used in the tuning process can be regarded as an *experimental unit* on which the observations can be obtained.

If the null hypothesis is rejected, post-Friedman tests are performed, comparing the best configuration to the others. Let the best candidate be referred to as θ_{best}. A candidate θ_j is considered different from the best one if:

$$\frac{|R_{\theta_{best}} - R_{\theta_j}|}{\sqrt{\frac{2K_{min}[(\sum_{j=1}^{K_{min}}\sum_{i=1}^{m} R_{ji}^2) - (\sum_{c=1}^{m} R_c^2)]}{(K_{min}-1)(m-1)}}} > t_{((1-\alpha/2),(K_{min}-1)(m-1))} \tag{2.4}$$

where

R_{ji} = the ranking of a candidate configuration i on an instance j
R_c = is the sum of rankings of a candidate configuration c on all K_{min} instances
$t_{((1-\alpha/2),(k_{min}-1)(m-1))}$ = is the $1-\alpha/2$ quantile of the Student's t distribution with

$(K_{min} - 1)(m - 1)$ degrees of freedom.

The race proceeds by evaluating the remaining candidate configurations on more instances, so as to iteratively increase the statistical power of the procedure and enable the differentiation of configurations that present smaller differences in performance. This procedure continues until either all but one configuration are discarded by the statistical testing, a given number of instances have been sampled, or a predefined computational budget has been exhausted.

Improvements to F-Race were proposed by Balaprakash *et al.* [37] in a method called **iterated F-Race** (I/F-Race). I/F-Race works by iteratively applying the F-Race procedure, generating new candidate configurations at each iteration by sampling from a multivariate random distribution of parameter values that is biased by the best configurations returned in the previous iterations. This biased sampling favours the search process for candidate configurations that are similar to the previous best, driving the search.

At the first iteration, a number of candidate configurations are sampled from an initial probabilistic model (e.g., a multivariate uniform distribution) and the F-Race is performed for these configurations. Based on the best candidate configurations returned, the sampling distribution of parameter values is adapted and new candidate configurations are sampled. The F-Race is then re-applied using the surviving configurations from the previous iteration plus the newly sampled ones, and this process is repeated until a termination criteria is reached.

Within each iteration of I/F-Race (i.e., for each race), two parameters regulate the number of statistical tests performed: K_{min} and K_{each}. As mentioned earlier in this section, parameter K_{min} defines how many instances must be sampled before the first statistical test is performed. Parameter K_{each} defines how many instances are sampled between tests, that is, how much does the sample size of the tests increase between tests. The total number of tests within each race depends on the computational budget allocated for the tuning procedure. The general framework of I/F-Race is given in Algorithm 1, adapted from [36].

I/F-Race requires an estimation of the number of iterations N^{iter}. The default setting is given by $N^{iter} = \lfloor 2 + log_2 N^{param} \rfloor$, where N^{param} expresses the number of parameters to be tuned. Given a total computational budget B, each iteration j performs one race with a limited computational budget $B_j = (B - B_{used})/(N^{iter} - j + 1)$, where $j = 1, ..., N^{iter}$. The number of candidate configurations N_j generated at each iteration j is $N_j = \lfloor B_j/(n + min(5, j)) \rfloor$. The number of candidates decreases with the number of

Algorithm 1 I/F-Race algorithm

Require: Search space (Ω); Instance Set (Γ); Tuning budget (B).
1: $t \leftarrow 0$ ▷ Initial Iteration
2: $N_{iter} \leftarrow$ CalculateIterations(Ω) ▷ Calculate the number of iterations
3: $B_{used} \leftarrow 0$ ▷ Initialize budget used
4: $B_{(0)} \leftarrow$ CalculateBudget(B, B_{used}, N_{iter}) ▷ Budget for Initial Iteration
5: $\mathcal{E}^{(t)} \leftarrow \emptyset$ ▷ Initialize the Elite Set
6: $\mathcal{A}^{(t)} \leftarrow$ SampleUniformConfs(Ω) ▷ Sample Initial Configs.
7: $\mathcal{E}^{(t)} \leftarrow$ Race($\mathcal{A}^{(t)}$,$B_{(0)}$, Γ) ▷ Performs a Race
8: Update(B_{used}) ▷ Update the budget used
9: **for** `t=1 to (`N_{iter}`-1)` **do**
10: $B_{iter}^{(t)} \leftarrow$ CalculateBudget(B, B_{used}, N_{iter})
11: $\mathcal{C}^{(t)} \leftarrow$ SampleNewConfs(Ω,$\mathcal{E}^{(t-1)}$) ▷ Gen. New Configurations
12: $\mathcal{A}^{(t)} \leftarrow \mathcal{A}^{(t-1)} \cup \mathcal{C}^{(t)}$
13: $\mathcal{E}^{(t)} \leftarrow$ Race($\mathcal{A}^{(t)}$,$B_{iter}^{(t)}$, Γ)
14: Update(B_{used})
15: **end for**
16: $\mathcal{P}_{\mathcal{E}}^{(t)} \leftarrow$ RetrieveElitePerformances$\left(\mathcal{E}^{(t)}\right)$
17: **return** Elite configurations ($\mathcal{E}^{(t)}$) and their estimated performance).

iterations. The parameter n allows the user to influence the denominator of the equation which determines N_j.

At the beginning of each new iteration j, a number of $N_j^{new} = N_j - N_{j-1}^{elite}$ new candidate configurations are generated. For generating a new candidate configuration, first one parent configuration θ_z is sampled from the set of elite configurations (Θ^{elite}) with a probability:

$$p_z = \frac{N_{j-1}^{elite} - r_z + 1}{N_{j-1}^{elite} \cdot (N_{j-1}^{elite} + 1)/2} \tag{2.5}$$

where r_z is the ranking of the candidate configuration θ_z. According to the equation 2.5, the candidate configurations with better ranking (lowest) have higher probability of being selected as parents. Once the parent configuration is selected, a new value is sampled for each parameter X_d, $d = 1, ..., N^{param}$. If X_d is a numerical parameter defined within the range $[\underline{x}_d, \overline{x}_d]$, then a new value is sampled from the truncated normal distribution $N(x_d^z, \sigma_d^j)$. The mean of the distribution x_d^z is the value of parameter d in elite configuration θ_z. The parameter σ_d^j is initially set to $(\overline{x}_d - \underline{x}_d)/2$, and it is decreased at each iteration before sampling:

$$\sigma_d^j = \sigma_d^{j-1} \cdot \left(\frac{1}{N_j^{new}}\right)^{1/N^{param}} \tag{2.6}$$

If X_d is a categorical parameter, with levels $X_d \in \{x_1, x_2, ..., x_{nd}\}$, then a new value is sampled from a discrete probability distribution $P^{j,z}(X_d)$. In the first iteration, $P^{j,z}(X_d)$

is uniformly distributed over the domain of X_d. In subsequent iterations, it is updated before sampling as follows:

$$P^{j,z}(X_d = x_j) = P^{j-1,z}(X_d = x_j) \cdot (1 - \frac{j-1}{N^{iter}}) + \Delta P \tag{2.7}$$

$$\Delta P = \begin{cases} \frac{j-1}{N^{iter}} \text{ if } x_j = x_z \\ 0 \text{ otherwise} \end{cases} \tag{2.8}$$

The new configurations generated after sampling inherit the probability distributions from their parents, and a new race is started with the new candidate configurations generated and those belonging to the elite configurations.

The I/F-Race has been widely used in a variety of works since its introduction in 2007, and is currently considered one of the most powerful general-purpose parameter tuning approaches for metaheuristics.

2.3.2 SMBO Methods

Methods based on SMBO (Sequential Model Based Optimization) for solving the *algorithm configuration problem* have their behavior inherited from the statistics literature, more specifically from blackbox optimization methods. From an initial set of data, these methods build a prediction model that represents a response surface which tries to explain the mathematical nature of the blackbox function. This response surface is optimized, generating new data which are used in the next iteration for generating a better response surface related to the process which is investigated. As the iterations progress, the response surfaces generated drive the method to find out good points located in local optima regions of the blackbox functions, maybe the global optimum. Some methods based on this principle are presented in the next sections.

2.3.2.1 Sequential Parameter Optimization (SPO)

The SPO (Sequential Parameter Optimization), which was proposed by Bartz-Beielstein *et. al* ([40]) is based on an approach of iteratively improving a statistical prediction model that tries to reveal the relation between algorithm parameters values and performance values obtained, and thus chooses the best parameter values. Its implementation, available both for Matlab and R languages is called **SPOT** - Sequential Parameter Optimization Tool [48].

Before starting the tuning itself, some information must be provided by the user to SPOT: the range of the parameter values; specific parameters used by the algorithm to be tuned, such as objective function, initial seed, starting point, etc; and specific parameters of SPOT, such as the prediction model, etc. Using these information, SPOT generates the experimental design in order to begin the iterations.

At the first iteration, the initial candidate configurations $\{\theta_1, ..., \theta_n\}$ are generated by using a LHS (Latin Hypercube Sampling) technique. These candidate configurations are evaluated on the instance a certain number of times and then a statistical prediction model F_0 is fitted. Based on the best candidate so far and F_0, a set of expected good new candidates are sampled. In short, predicted performance of new points based on F_0 and the modeling error are taken into account to estimate the probability of new candidates of being better than the known best so far, by a technique known as regression-kriging [48]. Those new points with highest probability of being better than the best are chosen. Then the best point so far and the new ones are run on the instance in a double times which was used to generate F_0. Afterwards, a new model F_1 is generated. Then, a new cycle is started, with a new group of candidate configurations obtained based on the new best candidate and the new model. At each cycle, the number of evaluations of the best candidate configuration and the news on the instance is doubled, obtaining a more accurate average candidate performance. These cycles continue until some termination criteria is reached. The standard initial prediction model used by SPOT is a second-order multiple linear regression model. The algorithm 2 describes more formally the SPO method.

The search for better configurations along the cycles is based on the successive refinements of the model, when the model relevance and the relevance of each parameter value are analyzed throught statistical tests, like ANOVA and t-tests. Other prediction model available for SPOT is regression trees. Another interesting aspect of SPOT is that, in a manner similar to REVAC (section 2.3.3.1), it stores information that allows the user to analyze the algorithm behavior regarding its parameters.

According to [48], SPOT was successfully applied to several optimization algorithms, specially in the field of evolutionary computation and for a variety of problems (machine engineering, aerospace industry, technical thermodynamics, bioinformatics, etc), obtaining better values than those found manually. However, an open question is about which models are preferable. Their authors argue that is unclear when a classical linear regression model or stochastic process models should be used.

Algorithm 2 SPO algorithm

Require: Search space (Ω); Instance used (γ); specific parameters of the algorithm. (P_{alg}); specific parameters of SPO. (P_{spo}).

1: $t \leftarrow 0$
2: $\mathcal{A}^{(t)} \leftarrow$ GenerateInitialSample (Ω) ▷ Sample initial configurations
3: $\mathcal{P}_{\mathcal{A}}^{(t)} \leftarrow$ EvaluateConfigurations $\left(\mathcal{A}^{(t)}, \gamma\right)$ ▷ Evaluate configs
4: $\mathcal{S}^{(t)} \leftarrow$ FitModel$\left(\mathcal{A}^{(t)}, \mathcal{P}_{\mathcal{A}}^{(t)}\right)$ ▷ Fit model
5: $\mathcal{E}^{(t)} \leftarrow \mathcal{A}^{(t)}$ ▷ Initialise elite archive
6: $\mathcal{C}^{(t)} \leftarrow$ GenerateNewConfigurations$\left(S^{(t)}, \mathcal{E}^{(t)}\right)$ ▷ Find new configurations
7: **while** Stop criteria not met **do**
8: $t \leftarrow t + 1$
9: $\mathcal{A}^{(t)} \leftarrow \mathcal{A}^{(t-1)} \bigcup \mathcal{C}^{(t-1)}$ ▷ Add candidate configs to archive
10: $\mathcal{P}' \leftarrow$ EvaluateConfigurationsDouble $\left(\mathcal{E}^{(t-1)}, \mathcal{C}^{(t-1)}, \gamma\right)$ ▷ Eval. elite and new configs 2 times
11: $\mathcal{P}_{\mathcal{A}}^{(t)} \leftarrow$ Update $\left(\mathcal{P}_{\mathcal{A}}^{(t-1)}, \mathcal{P}'\right)$ ▷ Update archive of config. performances
12: $\mathcal{S}^{(t)} \leftarrow$ FitModel$\left(\mathcal{A}^{(t)}, \mathcal{P}_{\mathcal{A}}^{(t)}\right)$ ▷ Fit model
13: $\mathcal{E}^{(t)} \leftarrow$ SelectKBest $\left(\mathcal{A}^{(t)}, \mathcal{P}_{\mathcal{A}}^{(t)}, K\right)$ ▷ Update elite archive
14: $\mathcal{C}^{(t)} \leftarrow$ GenerateNewConfigurations$\left(S^{(t)}, \mathcal{E}^{(t)}\right)$ ▷ Find new configurations
15: **end while**
16: $\mathcal{P}_{\mathcal{E}}^{(t)} \leftarrow$ RetrieveElitePerformances$\left(\mathcal{E}^{(t)}, P_{\mathcal{A}}^{(t)}\right)$
17: **return** Elite configurations ($\mathcal{E}^{(t)}$), their estimated performance ($\mathcal{P}_{\mathcal{E}}^{(t)}$) and the final model ($\mathcal{S}^{(t)}$).

2.3.2.2 BONESA

Proposed by Eiben and Smit [30], **Bonesa** is a tuning method based on two cooperative modules: a **Learning Loop** and a **Searching Loop**. They continuously exchange information during the iterations of tuning process, with the Learning Loop being responsible for using a prediction model for comparing candidate configurations, whereas the Searching Loop is responsible for sampling new candidate configurations, based on the candidate configurations previously compared by the Learning loop. This sampling is done in a way that the search is guided iteratively towards more promising candidate configurations. The most distinguishing difference between the other SMBO-based methods and Bonesa is its multi-objective approach. To be able to select the best parameter-values for completely different problems, Bonesa uses a Pareto-strength approach to optimize on a whole range of different problems in one go. The iteration between its two modules, as well as the multi-objective approach are detailed below.

To explain the method and its modules, let the candidate configurations be l-dimensional vectors θ. At the first iteration the **Searching Loop** generates k initial candidate configurations randomly, evaluating them once at each problem $\gamma \in P$ (set of all available problems) and storing them and their utility values (objective function values) in an archive A.

These candidate configurations in the archive A are used by the **Learning Loop**. For predicting values for unknown candidate configurations, an approach based on weighted average of the utilities of the closest known neighbors is used. The first step of this approach consists of determining the *standardized distance* between all pairs (θ_x, θ_y) of candidate configurations in A. A *standardized distance* between two l-dimensional vectors θ_x and θ_y is given by:

$$r(\theta_x, \theta_y) = \sqrt{1/l \cdot \sum_{i=1}^{l} ((\theta_{xi} - \theta_{y_i})/\overline{\sigma}_i)^2} \tag{2.9}$$

where: $\overline{\sigma}_i$ is the standard deviation of the parameter i of all "good" candidate configurations. Here, "good" candidate configurations are defined as those that perform above average on at least one of the problems. The closer two candidate configurations θ_x and θ_y are, lower is $r(\theta_x, \theta_y)$.

As the accuracy measured of the utility $\mu_{(\theta_x,\gamma)}$ of a candidate configuration θ_x on a problem γ is important for predicting correctly the parameter values for new candidate configurations, the noise must be minimized. Since each candidate configuration θ_x is evaluated once on each problem γ, for reducing noise, the estimated utility $\hat{\mu}_{(\theta_x,\gamma)}$ is given by the weighted average of the test results μ_γ of all candidates (including θ_x itself) in the same problem γ:

$$\hat{\mu}_{(\theta_x,\gamma)} = (\sum_{\theta_y \in A} \mu_{(\theta_y,\gamma)} \cdot \omega_{\theta_x,\theta_y})/(\sum_{\theta_y \in A} \omega_{\theta_x,\theta_y}), \text{ where} \tag{2.10}$$

$$\omega_{\theta_x,\theta_y} = e^{c \cdot r(\theta_x,\theta_y)}, \ \forall \ \theta_y \in A \tag{2.11}$$

The term c is a scaling factor, chosen in such a way that the average value of $\sum_{\theta_y} \omega_{\theta_x,\theta_y}$ is equal to 50, if the $\mid A \mid$ candidate configurations are uniformly distributed in a l-dimensional hypercube. Thus, candidate configurations far from a given candidate θ_x have smaller contributions to the estimated utility of θ_x than those near θ_x.

Given the equation above, the estimated variance $\hat{\sigma^2}_{(\theta_x,\gamma)}$ of a candidate configuration $\theta_x \in A$ on problem γ is:

$$\hat{\sigma^2}_{(\theta_x,\gamma)} = (\sum_{\theta_y \in A} (\mu_{(\theta_y,\gamma)} - \hat{\mu}_{(\theta_x,\gamma)})^2 \omega_{\theta_x,\theta_y})/((\sum_{\theta_y \in A} \omega_{\theta_x,\theta_y}) - 1) \tag{2.12}$$

The support $\hat{\rho}$ of a candidate configuration $\theta_x \in A$ is:

$$\rho\hat{\theta}_x = \sum_{\theta_y \in A} \omega_{\theta_x,\theta_y} \tag{2.13}$$

The equations 2.9 to 2.13 can be used to predict the utility, variance, and support of unseen candidate configurations. These measures can be derived using Gaussian interpolation over all candidate configurations in the file.

Once the utility of candidate configurations are predicted, they must be compared. Since BONESA was proposed for tuning multiobjective metaheuristics, the criteria used for comparing different candidate configurations is based on the principle of Pareto dominance. Considering a problem γ, and candidate configurations θ_z and θ_y, an adaptation of Welch's T-test is used to determine if the average utility of the candidate configuration θ_z is higher than the average utility of the candidate configuration θ_y on the problem γ. The notation 1-$D_{(\theta_z,\theta_y,\gamma)}$ indicates the level of confidence in the hypothesis that the average value of utility distribution of θ_z on problem γ is better or equal than the average value of the utility distribution of θ_y on γ. If this confidence is higher than a threshold $1-\epsilon$, θ_z is considered better than θ_y on γ. In relation to all problems $\gamma \in P$, a candidate configuration θ_z dominates θ_y if and only if:

1 $\exists \gamma \in P : D_{(\theta_z,\theta_y,\gamma)} \leq \epsilon$, and

2 $\forall g (g \neq \gamma) \in P : D_{(\theta_y,\theta_z,\gamma)} > \epsilon$.

The above statements 1 and 2 mean that: considering the set P of problems, there is at least one problem γ belonging to P for which θ_z has the average utility better than that of θ_y; and there is no problem belonging to P for which θ_y has the average utility better than that of θ_z.

Finally, is determined the Pareto Strength $Q(\theta_z)$ of a new generated candidate configuration θ_z, as:

$$Q(\theta_z) = \sum_{\theta_y \in A} \begin{cases} -1 \cdot S(\theta_y) \text{ if } \theta_z \text{ is dominated by } \theta_y \\ 0 \text{ if } \theta_z \text{ is not dominated by } \theta_y \end{cases} \tag{2.14}$$

where $S(\theta_y)$ is given by:

$$S(\theta_z) = \sum_{\theta_y \in A} \begin{cases} 1 \text{ if } \theta_z \text{ dominates } \theta_y \\ 0 \text{ if } \theta_z \text{ does not dominate } \theta_y \end{cases} \tag{2.15}$$

At this point, a new set of candidate configurations is generated by the **Searching Loop**. This task is made in two steps: 1) for each one of the parameters ($i = 1, ..., l$) of each candidate configuration a value x_i is randomly drawn from the corresponding parameter values in the archive A. 2) for each parameter value x_i, is added a Gaussian noise, in a way that the new value is expected to be within the top 50 closest values. To this end, the level of noise s_i is defined as follows:

$$s_i = \sqrt{12} \cdot \overline{\sigma_i} \cdot (\Gamma(0.5 \cdot l + 1) \cdot (50 \cdot 1/ \mid A \mid))^{1/l} / \sqrt{\pi \cdot \mid A \mid} \tag{2.16}$$

where $\overline{\sigma}_i$ is the standard deviation of the parameter i in file A and Γ is the gamma function. This process is repeated until K new candidate configurations are created. For each of these, the Pareto strength is calculated by **Learning Loop**. Next, the m candidate configurations with highest Pareto Strength are selected, tested on each of problems by the **Searching Loop** and added to the file A.

At the following iterations, the process of comparing candidate configurations by **Learning Loop** and generating new K candidates configurations and new m best candidate configurations by **Searching Loop** is repeated until a maximum number of tests is reached. Increasing K or lowering m BONESA can be made more exploratory or exploitative. The algorithm of Bonesa is descript as follows (based on [49]).

Algorithm 3 Bonesa algorithm

Require: Search space (Ω); Set of instances (Γ); number of initial configs. (m_0); tuning budget (B); number of new configurations K).

1: $\mathcal{A} \leftarrow$ GenerateInitialSample (Ω, m_0) ▷ Sample initial configurations
2: $\mathcal{P}_{\mathcal{A}} \leftarrow$ EvaluateConfigurations $(\mathcal{A}, \Gamma)$ ▷ Evaluate configs. on instances
3: $i \leftarrow m_0$
4: **while** maximum budget B not reached **do**
5: $\mathcal{M}_d \leftarrow$ ModelDensityCandidates($\mathcal{A}$) ▷ Build a Model of Density distribution of configs.
6: $\mathcal{M}_p \leftarrow$ ModelPredQuality($\mathcal{A}, \Gamma$) ▷ Build a Model of predict performance configs.
7: $\mathcal{M}_v \leftarrow$ ModelPredVar($\mathcal{A}, \Gamma$) ▷ Build a Model of predict performance variance configs.
8: $\mathcal{C} \leftarrow \emptyset$ ▷ Initialize set of new configurations
9: **for** `j=1 to` K **do** ▷ Generating new coonfigurations
10: $\theta^j_{par} \leftarrow$ DrawConfig($\mathcal{M}_d$) ▷ Sample parameter values of the j-th new configuration
11: $\theta^j_p \leftarrow$ PredPerf$\left(\mathcal{M}_p, \theta^j_{par}, \Gamma\right)$ ▷ Predict the performance of the j-th configuration
12: $\theta^j_v \leftarrow$ PredVar$\left(\mathcal{M}_v, \theta^j_p, \Gamma\right)$ ▷ predict the variance of the j-th configuration
13: $\theta^j_r \leftarrow$ PredParRank$\left(\theta^j, \mathcal{A}\right)$ ▷ Predict the pareto rank of the j-th configuration
14: $\mathcal{C} \leftarrow \mathcal{C} \cup \theta^j$ ▷ Update the set of new configurations
15: **end for**
16: $\theta^* \leftarrow$ GetBest($\mathcal{C}$)
17: $\theta^*_p \leftarrow$ EvaluateConfiguration(θ^*, Γ) ▷ Obtain the actual performance of θ^*
18: $\mathcal{A} \leftarrow \mathcal{A} \cup \theta^*$
19: **end while**
20: $\mathcal{R} \leftarrow$ CalculateParetoRank$(\mathcal{A}, \Gamma)$
21: $\mathcal{E} \leftarrow$ GetNondominated$(\mathcal{R}, \mathcal{A})$
22: **return** Non dominated configuratioins ($\mathcal{E}$).

BONESA was applied in a combination of an evolutionary algorithm and a class of problems for which the complete performance landscape was known. Actually, this is an artificial controlled perfomance landscape. Using two parameters, the experimental results obtained by BONESA was very close to those real landscape performances, mainly for the values of parameters for which the highest performances were reached.

2.3.2.3 SMAC

Proposed by Hutter, Hoos and Brown [50], SMAC (Sequential Model-Based Algorithm Configuration) was designed for optimizing blackbox functions that arise in the optimization of algorithm parameters. Given an initial set of candidate configurations $\{\theta_1, \ldots, \theta_m\} \in \Theta$ (the set of all possible candidate configurations) and their objective function values $\{f(\theta_1), \ldots, f(\theta_m)\}$, the method iterates over the following steps [50]:

(1) based on the initial data collected, construct a model which predicts the objective function value $f(\theta)$ for an arbitrary candidate configuration θ;

(2) Using the model, performs a multi-start search for finding the candidate configuration θ^*, that maximizes the *expected positive improvement function* $E[I(\theta)] = E[max\{0, f_{min} - f(\theta)\}]$, where f_{min} is the minimum value of $f(\theta)$ gathered so far, and $f(\theta)$ is a prediction objective function value for an arbitrary θ, according to the model used;

(3) Evaluate $f(\theta^*)$ and add it to the pool of points (candidate configurations)

The algorithm 4 describes the main steps of SMAC. In the work presented in [50], in order to apply SMAC specifically to the BBOB set of blackbox functions, the authors made two modifications in it: firstly, the SMAC-BBOB version used GP (Gaussian Process) models instead of Random Forests based models. According to the authors, GP models improve the performance for continuous optimization. Secondly, SMAC-BBOB used two approachs of multi-start search instead one: it used DIRECT algorithm (proposed by [51] *apud* [50]) and CMA-ES algorithm (proposed by [52] *apud* [50]) in order to intensify the search using model evaluation. According to the authors, SMAC-BBOB in most cases outperformed the state-of-art blackbox optimizer CMA-ES. However, CMA-ES presented better results in a presence of larger budgets, like 100 x D function evaluations, where D is the dimension of functions.

In [41], Hutter, Hoos and Brown modified the SMAC algorithm to deal with categorical parameters and sets of instances, instead of single instances. This extended version used a model based on Random Forests and includes in this model information about

Algorithm 4 SMAC algorithm

Require: Search space (Ω); instances (Γ_S); number of initial configs. (m_0);number of addit. instances/iter. ($N_\star$);).

$t \leftarrow 0$
$\mathcal{A}^{(t)} \leftarrow$ GenerateInitialSample (Ω, m_0) ▷ Sample initial configurations
$\Gamma_{\mathcal{A}}^{(t)} \leftarrow$ Sample$(\Gamma_S, N_0 - N_\star)$ ▷ Sample initial instances
$\mathcal{P}_{\mathcal{A}}^{(t)} \leftarrow$ EvaluateConfigurations $\left(\mathcal{A}^{(t)}, \Gamma_A^{(t)}\right)$ ▷ Evaluate configs on instances
$\mathcal{S}^{(t)} \leftarrow$ FitModel$\left(\mathcal{A}^{(t)}, \mathcal{P}_{\mathcal{A}}^{(t)}\right)$ ▷ Fit model
$\mathcal{E}^{(t)} \leftarrow \mathcal{A}^{(t)}$ ▷ Initialise elite archive
$\mathcal{C}^{(t)} \leftarrow$ GenerateNewConfigurations$\left(S^{(t)}, \mathcal{E}^{(t)}\right)$ ▷ Find new configurations
while Stop criteria not met **do**
  $t \leftarrow t + 1$
  $\mathcal{A}^{(t)} \leftarrow \mathcal{A}^{(t-1)} \bigcup \mathcal{C}^{(t-1)}$ ▷ Add candidate configs to archive
  $\Gamma' \leftarrow$ Sample $\left(\Gamma_S \backslash \Gamma_{\mathcal{A}}^{(t-1)}, N_\star\right)$ ▷ Sample new instances
  $\mathcal{P}' \leftarrow$ EvaluateConfigurations $\left(\mathcal{A}^{(t)}, \Gamma'\right)$ ▷ Eval. elite configs on new instances
  $\Gamma_{\mathcal{A}}^{(t)} \leftarrow \Gamma_{\mathcal{A}}^{(t-1)} \bigcup \Gamma'$ ▷ Update archive of instances visited
  $\mathcal{P}_{\mathcal{A}}^{(t)} \leftarrow$ Update $\left(\mathcal{P}_{\mathcal{A}}^{(t-1)}, \mathcal{P}'\right)$ ▷ Update archive of config. performances
  $\mathcal{S}^{(t)} \leftarrow$ FitModel$\left(\mathcal{A}^{(t)}, \mathcal{P}_{\mathcal{A}}^{(t)}\right)$ ▷ Fit model
  $\mathcal{E}^{(t)} \leftarrow$ SelectKBest $\left(\mathcal{A}^{(t)}, \mathcal{P}_{\mathcal{A}}^{(t)}, K\right)$ ▷ Update elite archive
  $\mathcal{C}^{(t)} \leftarrow$ GenerateNewConfigurations$\left(S^{(t)}, \mathcal{E}^{(t)}\right)$ ▷ Find new configurations
end while
$\mathcal{P}_{\mathcal{E}}^{(t)} \leftarrow$ RetrieveElitePerformances$\left(\mathcal{E}^{(t)}, P_{\mathcal{A}}^{(t)}\right)$
return Elite configurations ($\mathcal{E}^{(t)}$), their estimated performance ($\mathcal{P}_{\mathcal{E}}^{(t)}$) and the final model ($\mathcal{S}^{(t)}$).

categorical parameters and features of the instances used. Thus, the model was able to predict the performance of an arbitrary candidate configuration θ, containing numerical and categorical parameters on a set of instances $\{i_1, \ldots, i_q\}$ about which some features are known by the model. In the following step, this model was used for generating new promising candidate configurations that are added to the set of candidate configurations used for generating the next model.

Some new candidate configurations were generated based on the neighborhood of a group of the k best ones, according to the values of the measure *expected positive improvement function* $E[I(\theta)]$; and others were generated randomly. One aspect not observed by the authors of using Random Forests is that, like Ruth and Loughin pointed out ([53]), in the presence of heterocedastic data the trees which form a Random Forest can determine wrong locations and number of splits on the data, resulting in a model whose capability of prediction is damaged. As there is no guarantee that the algorithm performance data are always homocedastic, it can be a problem.

Computational experiments showed that SMAC is a robust tuning method. It was compared with some other 4 methods, among them SPO (section 2.3.2.1) and ParamILS

(section 2.3.3.2). The performance of the methods was observed when tuning 3 solvers (SAPS, SPEAR and IBM ILOG CPLEX) for single and multiple instances, in 17 different configuration scenarios (each configuration scenario defines a different set of parameters to be tuned and different instances to be used). In order to analyze the performance of all methods, the best configurations found by each method were run 25 times on each scenario, the average performance gathered and differences among them investigated statistically.

Considering the case of a single instance, 11 different scenarios were used for tuning the solvers SAPS and SPEAR and each method was run during 30 minutes on each scenario. The results obtained by the best configurations found by the methods showed that for scenarios containing a few number of parameters SMAC and SPOT had the best performances. For scenarios with large number of parameters SMAC achieved the best results.

In terms of multiple instances, 6 different scenarios were used for tuning the solvers SAPS, SPEAR and CPLEX. Each method was run during 5 hours on each scenario. In this case, SMAC achieved the best results in all 6 scenarios.

2.3.3 Hyperheuristics

The term *hyperheuristics* [54–57] is used here to classify those tuning methods which consist in the application of metaheuristics for obtaining the best parameter values of algorithms, trying to solve the parameter tuning problem by directly tackling its optimization formulation, discussed in Section 2.2. While in principle any optimization approach could be used to solve the parameter tuning problem, knowledge about the characteristics of this problem have motivated the development of specific strategies. Three of the most common ones are REVAC [26, 58], ParamILS [35] and CRS-Tuning [42], as presented below.

2.3.3.1 REVAC

Nannen and Eiben [26] proposed a tuning method for Evolutionary Algorithms (EA) that aims to answer two questions: *(i) from the set of parameters, which of them are in fact relevant ?, that is, which of them have an effective influence to the performance of the EA*; and *(ii) For those relevant parameters, what are their best values, that is, the values which leads the algorithm to the best results?*. Considering these two questions may be important, since a considerable number of parameters can reveal themselves irrelevant for the performance of EA, being discarded from its original design.

The REVAC is an evolutionary strategy itself. Considering an EA with k parameters $\{X^1, ...X^k\}$, REVAC begins with a population of candidate configurations $P_1 = \{\theta_1, ..., \theta_n\}$, each of them represented by a vector $\theta_i = \{x^1, ...x^k\}$ of parameter values. Each parameter value $\{x^1, ...x^k\}$ of each candidate configuration θ_i is drawn randomly from the domains $\{D^1, ...D^k\}$ of the parameters $\{X^1, ...X^k\}$. In the context of the EA, each candidate configuration can be seen as a version of the algorithm whose parameters must be tuned. Each candidate configuration is evaluated according to an objective function $f(\theta_i)$, generating a response surface $r = f(\theta_i)$.

The next generations of candidate configurations are obtained using operators of recombination and mutation which are applied over the parameter values of the candidate configurations, implementing the search process for good parameter values. About the relevance of parameters, at each generation t, k marginal density functions $M_t = \{D_t^1, ..., D_t^k\}$ (each one corresponding to the values to be sampled for each parameter of EA) are built from the old set $M_{t-1} = \{D_{t-1}^1, ..., D_{t-1}^k\}$ of the prior iteration. In this way, new distributions of the parameter values are built on estimates of the response surface that were sampled with previous density functions, giving, at each iteration, a higher probability to regions of response surface with better results. The concept of Shannon entropy (Shannon *apud* Nannen [26]) of these distributions is used to estimate the relevance of the parameters, measuring how this information is distributed over them. The algorithm of revac is presented as follows (adapted from [59]):

Algorithm 5 REVAC algorithm

Require: Search space (Ω); Instance used (γ); the Population size (m), Number of parent configurations (K); Number of generations (G_{max})

1: $G \leftarrow 0$
2: $\mathcal{A}^{(G)} \leftarrow$ GenerateInitialSample (Ω) ▷ Sample initial configurations
3: $\mathcal{P}_{\mathcal{A}}^{(G)} \leftarrow$ EvaluateConfigurations $\left(\mathcal{A}^{(G)}, \gamma\right)$ ▷ Evaluate configs
4: $\mathcal{P}^{(G)} \leftarrow$ SelectKParents $\left(\mathcal{A}^{(G)}, \mathcal{P}_{\mathcal{A}}^{(G)}, K\right)$ ▷ Update elite archive
5: **while** not(G_{max}) **do**
6: $G \leftarrow G + 1$
7: $\mathcal{C}^{(G)} \leftarrow UniformCrossover(\mathcal{P}^{(G-1)})$ ▷ Crossover operation
8: $\mathcal{C}^{(G)} \leftarrow Mutation(\mathcal{C}^{(G)})$ ▷ Mutation operation
9: $\mathcal{P}' \leftarrow$ EvaluateChildConfs $\left(\mathcal{C}^{(G)}, \gamma\right)$ ▷ Eval. Child Configurations
10: $\mathcal{A}^{(G)} \leftarrow$ ReplaceOldestConfs$\left(\mathcal{A}^{(G-1)}, \mathcal{C}^{(G)}\right)$ ▷ Replace Oldest configs. by child
11: $\mathcal{P}_{\mathcal{A}}^{(G)} \leftarrow$ Update $\left(\mathcal{P}_{\mathcal{A}}^{(G-1)}, \mathcal{P}'\right)$ ▷ Update archive of config. performances
12: $\mathcal{P}^{(G)} \leftarrow$ SelectKParents $\left(\mathcal{A}^{(G)}, \mathcal{P}_{\mathcal{A}}^{(G)}, K\right)$ ▷ Update elite archive
13: $\mathcal{S}_{\Theta}^{(G)} \leftarrow$ CalcEntropy $\left(\mathcal{A}^{(G)}\right)$ ▷ Calc. the Shannon Entropy for the set Θ of parameters
14: **end while**
15: $\mathcal{E}^{(t)} \leftarrow$ RetrieveEliteConfigurations$\left(\mathcal{A}^{(G)}, \mathcal{P}_{\mathcal{A}}^{(G)}\right)$
16: **return** Elite configurations ($\mathcal{E}^{(t)}$) and the Shannon Entropy Set ($\mathcal{S}_{\Theta}^{(G)}$).

According to [26], for those parameters for which entropy decreases quickly along the generations, little information is needed to tuned them, and therefore they are more

relevant for the performance of the EA. On the other hand, for those that this decreasement of entropy does not happen, it means that they are less relevant, and thus they may be discarded. At the last generation, REVAC returns the parameters relevance and the best parameter values obtained by tuning.

REVAC was tested in simulated and real scenarios by tuning an EA, and its results were promising. In simulated tests, it showed to be able to find values of known parameters (recombination and mutation) close to those used in literature, and its results for relevance (mutation more relevant than recombination) was equivalent to the literature [26]. In an another experiment with an EA for solving incoming in economic environments, REVAC was tested in 18 simulated economic environments and it was able to reduce the original number of 13 to 6 parameters, due to irrelevance of 7 parameters in the performance of EA; besides to reach good values for the 6 relevant parameters [58].

2.3.3.2 ParamILS

The **ParamILS** is a framework of tuning methods proposed by Stutzle *et. al* [35] that is based on the idea of Iterated Local Search metaheuristic (**ILS**). Basically, starting from a candidate configuration θ selected from a group of initial candidate configurations sampled randomly, a first improvement local search is applied over θ, generating a θ_{ils} candidate. After this, a certain number of cycles of exploration and exploitation are performed, until some termination criteria is reached.

At each cycle, the best candidate θ_{ils} of the prior cycle is perturbed, generating an new candidate neighbor θ_p. The new candidate θ_p is submitted to a procedure of first improvement local search, generating a new candidate θ_l. After the local search, if θ_l is better than the initial candidate θ_{ils}, the current candidate θ_{ils} is replaced by θ_l, that is, $\theta_{ils} \leftarrow \theta_l$. Finally, the candidate θ_{ils} can be replaced by a candidate θ sampled randomly with a probability of $p_{restart}$, that is $\theta_{ils} \leftarrow \theta$ with a probability of $p_{restart}$ and a new cycle is started with the candidate θ_{ils}. The algorithm 2 describes the ParamILS (adapted from [35]).

The neighbourhood $Nbh(\theta)$ of a configuration θ is the set of all configurations that differ from θ in one parameter. The function *better* performs statistical tests in a blocking technique, after running the candidates on N instances.

The variants of the basic algorithm described are **FocusedILS** and **Adaptive Capping of Algorithm Runs** presented in the same original paper. The FocusedILS adaptively selects the number of training instances used by candidate configurations. It deals with this problem adaptively varying the number of training samples considered from one

Algorithm 6 ParamILS algorithm

Require: Search space (Ω); the initial candidate θ_0; initial random search r; random moves s for perturbation; $p_{restart}$ and perturbation index s

for $i = 1, ..., r$ **do**
  $\theta \leftarrow$ random $\theta \in \Theta$;
  if better(θ,θ_0) then $\theta_0 \leftarrow \theta$
end for
$\theta_{ils} \leftarrow$ IterativeFirstImprovement(θ_0)
while not TerminationCriteria() **do**
  $\theta \leftarrow \theta_{ils}$
  for $i = 1, ..., s$ **do** ▷ Perturbation
    $\theta \leftarrow$ random $\theta^{'} \in$ Nbh(θ);
  end for
  $\theta_l \leftarrow$ IterativeFirstImprovement(θ) ▷ Basic local search
  if better(θ_l, θ_{ils}) then $\theta_{ils} \leftarrow \theta_l$ ▷ AcceptanceCriterion
  with probability $p_{restart}$ do $\theta_{ils} \leftarrow$ random $\theta \in \Theta$
end while
return overall best θ found
Procedure IterativeFirstImprovement(θ)
repeat
  $\theta^{'} \leftarrow \theta$
  for each $\theta'' \in$ Nbh(θ') in randomized order **do**
    if better($\theta^{''}$,$\theta^{'}$) then $\theta \leftarrow \theta''$; break;
  end for
until $\theta' = \theta$
return θ

candidate configuration to another, giving priority to those with better performance along the evaluations. The adaptive capping of Algorithm Runs controls determines a runtime upper bound for each run of the candidates, based on the previous runtimes of the fastest candidate configuration, during the process of comparing two candidate configurations. This strategy avoids wasting much time for running worst candidates.

The three approaches were tested in [35] with 3 algorithms: SAPS (a high-performance dynamic local search algorithm for SAT problem), SPEAR (a recent three algorithm for solving SAT problems) and CPLEX (commercial tool CPLEX 10.1.1, a massively parameterized algorithm for solving mixed integer programming (MIP) problems). Three sets of benchmark instances were used: SAT-encoded quasi-group completion problems, SAT-encoded graph-colouring problems based on small graphs, and MIP-encoded winner determination problems for combinatorial actions.

The computational tests showed that all algorithms had improvement in their performance, comparing their default configuration and their best configuration determined by the methods **ParamILS** and **FocusedILS**. Two approaches were tested for sampling the initial candidates in **ParamILS**: a *random search* and *first improvement local search*, and the second approach presented the better results (3 bests results of 5 test scenarios). In a comparison between **ParamILS** and **FocusedILS** the second approach

presented better results (3 best results of 5 test scenarios). The **ParamILS** and **FocusedILS** were tested with Adaptive Capping and both of them had their performance improved. Finally, the CPLEX was testes in real-world Benchmarks (according to the authors, it was the first published work on automatically configuring the CPLEX mixed integer programming solver) , overcoming the results of standard configurations, for all benchmarks.

Engelke and Ewald [60] presented a fine-tuned simulation algorithm, using ParamILS. Specifically, they tuned the simulation framework JAMES II (Himmelspach and Uhrmacher *apud* Engelke and Ewald [60]). Their preliminary results indicated that ParamILS is suitable to find good parameter configurations of JAMES II. KhudaBukhsh et al. [61], presented a SLS (Stochastic Local Search) solver SATenstein for the propositional satisfiability problem (SAT). They used ParamILS for tuning SATenstein parameters and obtained new state-of-art results, when comparing its performance against 11 other SLS-based SAT solvers on 6 different instance groups available in literature.

2.3.3.3 CRS-Tuning

CRS-Tuning [42] is a tuning method for numerical and categorical parameters, which is composed of an evolutionary strategy combined with an approach called Chess Rating System (CRS), which is used to rank the configurations. In this method, initial configurations are randomly generated, and for each tuning instance each configuration is run n times. Candidate configurations are compared pairwise, and the results of these comparisons (in terms of wins, losses and draws) are used to calculate a rating R, as well as a rating deviation RD, and a rating interval RI for each configuration, which describe its relative qualities. Configurations that are considered as significantly worse than the best one are eliminated, and finally crossover and mutation operators are applied to the surviving configurations to create new ones, and the procedure iterates. The procedure is run until the maximum number of executions has been reached. The algorithm that describes this approach is the following (adapted from [42]):

CRS-Tuning was compared to Irace and REVAC for tuning parameters of three metaheuristics: Artificial Bee Colony, Differential Evolution and Gravitational Search Algorithm, when these algorithms were employed to solve eight classical challenging optimization problems. The results showed that CRS-Tuning was able to provide competitive configurations, when compared to those returned by Irace and REVAC; and that in general it found the best configurations faster than Irace.

Algorithm 7 CRS-Tuning algorithm

Require: Search space (Ω); Instances Set (Γ); the Population size (m), Number of parent configurations (K); Number of generations (G_{max}); Crossover probability Cr and Mutation probability F.)

$G \leftarrow 0$
$\mathcal{A}^{(G)} \leftarrow$ GenerateInitialSample (Ω) ▷ Sample initial configurations
while not(G_{max}) **do**
  $G \leftarrow G + 1$
  $\mathcal{P}_{\mathcal{A}}^{(G)} \leftarrow$ EvaluateConfigurations $\left(\mathcal{A}^{(G)}, \Gamma, ng\right)$ ▷ Evaluate configs in each instance a number of ng times
  $CRS^{(G)} \leftarrow$ CRSComparison($\mathcal{A}^{(G)}$, $\mathcal{P}^{(G)}$) ▷ Performs and store the CRS comparison
  $Rank^{(G)} \leftarrow$ RankCRSConfs$\left(CRS^{(G)}, \mathcal{A}^{(G)}\right)$ ▷ Rank configs. according CRS
  $Best^{(G)} \leftarrow$ SelectBest($Rank^{(G)}$)
  $\mathcal{A}^{(G)} \leftarrow$ EliminateWorst($\mathcal{A}^{(G)}, Best^{(G)}$, $Rank^{(G)}$) ▷ Eliminate worst than best
  $\mathcal{P}^{(G)} \leftarrow$ SelectKParents$\left(\mathcal{A}^{(G)}, K\right)$ ▷ (Select K Parents)
  while $|\mathcal{A}^{(G)}| < m$ **do**
    $\theta_{child} \leftarrow$GenerateNewConfig$\left(\mathcal{P}^{(G)}, Cr, F\right)$ ▷ Generate a child configuration
    $\mathcal{A}^{(G)} \leftarrow \mathcal{A}^{(G)} \cup \theta_{child}$ ▷ Update pool of configs.
  end while
end while
$\mathcal{E}^{(t)} \leftarrow$ RetrieveEliteConfigurations$\left(\mathcal{A}^{(G)}, \mathcal{P}_{\mathcal{A}}^{(G)}\right)$
return Elite configurations ($\mathcal{E}^{(t)}$)).

2.4 Comparing Tuning Algorithms

Smit [62] presents one comparison of 5 tuning methods: REVAC, SPO, Bonesa, SMAC and Irace. The target algorithm chosen to be tuned was the top of the BBOB'10 competition $(1, \lambda_m^s)$-CMA-ES (Auger, Brockhoff and Hansen *apud* Smit [62]), and six numeric (integer and real) parameters were tuned. Five functions from the BBOB'10 competition were selected as instances to be solved and the performances of the best parameter values found by the tuning methods were compared with those obtained by the standard parameter values used in the BBOB'10 competition. Here, the performance have to be understood as the number of function evaluations needed to reach a certain fitness threshold, with a maximum allowed number of evaluations of 20000. For each tuning method, for each problem, 10 best configurations were obtained and each one were run 100 times and the average performance considered. It was allowed to each tuning method to do 10000 experiments on each problem, that is, each one sampled 10000 results of running candidate configurations on each problem. Only Bonesa were analyzed under two angles: as a method for obtaining a best parameter vector for each problem (similarly to the others) and for obtaining a best parameter vector for all problems.

Considering each problem, Bonesa and SMAC obtained the best results both on performance and stability. In general, they had the best performance, and were able to reach this level of performance in most of the runs. Another interesting result was that for

two problems none of the tuners managed to find a parameter vector that performed significantly better than the recommended parameters used for the BBOB'10 competition. Considering a multi-problem approach, Bonesa was able to identify a candidate configuration which was better than the recommended parameter values in a half of the runs (5 runs) on each problem. Finally, they stated an ordering for the tuning methods on this specific test-suite: Bonesa and SMAC were the best, in a second group were SPO and Irace and the last was REVAC.

Monteiro et al. [59] presents a comparison of five parameter tuning methods: a *F-Race* based approach, REVAC, ParamILS, SMAC, and a Random Search based approach implemented by them. It was considered for tuning the following parameters of a standard genetic algorithm (SGA): the selection, crossover and mutation operators. Eight classical test functions was used in this experiment, being 4 as belonging to the instances training set, and 4 to the instances validation set. The fitness criterion of a SGA execution was the amount of evaluations to find the optimum (fixed as a given value for each function). Each execution of the SGA performed a maximum amount of 10^5 function evaluations, and when the optimum cannot be reached, the fitness of a SGA run was set as the maximum amount of function evaluations fixed.

For all parameter tuning methods, a budget of 4000 executions of the SGA was considered. Each parameter tuning method was run 5 times on the instances training set, and the performance of each candidate configuration returned by each method was measured after running it 100 times on each instance from the validation set using different random seeds, and counting the number of function evaluations required to reach the optimum values fixed. The results showed that is difficult to determine one method which is always the best for the task of tuning. Although surprisingly the average performance of the Random approach was comparable to the other methods, this approach was considerably more time consuming than the others.

After presenting the theoretical aspects of the Parameter Tuning Problem, as well as a brief review of the main works related to the parameter tuning methods, the next chapter presents the concepts used in this work to propose a new parameter tuning method.

Chapter 3

Proposed Tuning Framework

3.1 Introduction

As presented in the previous chapter, a variety of approaches can be found in the literature for solving the *Algorithm Configuration Problem*. Based on concepts present in several of these methods, this work proposes a parameter tuning framework based on the optimization of sequentially-generated response surfaces. These models are built to represent the desired typical behavior (mean, median, etc) of the algorithm configurations for a desired problem class, represented by a sample of tuning instances.

The analysis of response surfaces is used in many problems [63] in order to derive equations which explain approximately the behavior of a *response variable* (which can represents the average performance of a metaheuristic when solving a problem class - an Genetic Algorithm (GA) solving some instances of the Traveling Salesman Problem, for instance); according to the known values of some *input variables* (representing factors that lead to the average performances of the metaheuristic - values of the mutation rate and the crossover rate of the GA, for instance). This approximate knowledge about the behavior of the experiment according to the values of the *input variables* allows the experimenter to have more control about it, and in this way he/she can predict or optimize it, with a certain level of confidence.

To describe the proposed tuning method, following the formulation presented for the Algorithm Configuration Problem (Section 2.2), it is important to restate some definitions, as well as to present others:

- Θ is the set of all possible parameter configurations for a given metaheuristic.
- Γ is the set of all problem instances belonging to a certain problem class.

- γ_j represents one instance sampled from the set Γ
- θ_i represents one candidate configuration sampled from the set Θ
- each candidate configuration $\theta_i = \{v_1, \ldots, v_n\}$ is a vector of n parameter values. Each parameter value v_i is related to a parameter X_i. For each parameter X_i, its value v_i can be a value in the interval $[\underline{v}_i, \overline{v}_i]$. This interval represents the domain of X_i.
- $p_{\theta_i,\gamma_j} \sim \mathcal{P}(\theta_i, \gamma_j)$, where $\mathcal{P}(\theta_i, \gamma_j)$ represents the probability distribution function of performance values returned by applying the configuration θ_i on an instance γ_j.
- $\chi_{(\theta_i;\Gamma)}$ is the set of values $p_{\theta_i;\gamma}$ for all $\gamma \in \Gamma$, that is, the set of expected performances of a candidate configuration θ_i on all instances belonging to problem class Γ.
- $\mu_{(\theta_i;\Gamma)}$ is the expected value of $\chi_{(\theta_i;\Gamma)}$.

It is then possible to state the *algorithm configuration problem* as:

$$\text{Find } \theta^* = \arg\max_{\theta} \mu_{(\theta;\Gamma)} \tag{3.1}$$

that is, the problem of finding the configuration that maximizes the expected performance of a given metaheuristic for a given class of instances.

This chapter is organized as follows: first, an overview of the proposed tuning framework is provided, with an explanation of its main modules and the expected roles each one is expected to perform. After that, the specific aspects of the proposed implementation are provided, with a more detailed explanation of the design choices of each module.

3.2 Proposed Tuning Framework

The tuning framework proposed in this work can be seen as a SMBO method (2.3.2) based on the following tasks, according to the iterations:

At the first iteration:

1. Based on the set of candidate configurations Θ, sample the first m candidate configurations $\{\theta_1,\ldots,\theta_m\}$ and evaluate them on a set of the first q instances, $\{\gamma_1,\ldots,\gamma_q\}$, sampled from Γ.

2. Based on the observed values of p_{θ_i,γ_j} $(\forall i = 1 \ldots m, \quad \forall j = 1 \ldots q)$, build a set of t response surfaces representing likely mappings from the set of parameter values to a given statistic of the performance of the metaheuristic on the problem class Γ.

3. Optimize these t response surfaces, find a set of t best candidate configurations $\{\theta_1, \ldots, \theta_t\}$ representing the estimated best parameter configurations; and

4. Add the new t candidate configurations $\{\theta_1, \ldots, \theta_t\}$ to the pool of candidate configurations and evaluate them on the instances sampled so far.

5. Choose the best k candidate configurations, based on the performance of all candidate configurations on all instances sampled so far.

From iteration 2 onwards:

1. Sample new q' instances from Γ.

2. Evaluate the k best candidate configurations of the prior iteration on the new q' instances.

3. Based on all performance data of candidates on instances p_{θ_i,γ_j} $(\forall i = 1 \ldots m,$ $\forall j = 1 \ldots q)$, build a set of t response surfaces. In order to build these response surfaces, the overall performance of each candidate is weighted according to the number of instances the candidate was evaluated.

4. Repeat the steps 3 and 4 which are described for iteration 1.

5. Choose the best new k candidate configurations, based on the performance of the k best candidate configurations of the prior iteration and the new t candidates generated in the current iteration.

The steps described for iteration 2 are repeated until some termination criteria is reached, which is usually some form of exhaustion of the computational budget (time, iterations, runs, etc). Assuming that the number of instances available for the tuning effort at the r-th iteration, q_r, and the number of candidate evaluations evaluated until that iteration, m_r, provide the method with sufficient information to obtain response surfaces that are somehow representative of the main features of the average performance landscape of an algorithm on a given problem class; optimizing them will allow to the method to perform a search in different regions of the parameters space.

Moreover, adding the new candidates generated by optimizing the response surfaces to the pool of candidates and choosing only the best to further evaluations on new

instances will gradually drive the method to generate models that represent regions of the parameters space with the highest average performance. At the same time, using all performance data in a weighted way to build the models implies generating models with more prediction accuracy in those regions of the best candidates (those evaluated on more instances), but somehow represents the algorithm behavior across all parameters space explored, once the performance data of all candidates generated are used.

In the next sections we present a modular structure for tackling the parameter tuning problem presented in Section 2.2. The proposed framework, which we will refer to as *MetaTuner*, can be used to instantiate distinct tuning approaches through the adoption of specific methods for each of its components, depending on the nature of the tuning process at hand. This modular approach results not only in a greater flexibility for the framework, but is also useful for faster development and testing of proposed improvements. A more detailed explanation about the components of the method is given.

3.2.1 Initial Candidates

Considering the importance of gathering enough information for generating the first group of regression models, it is important to guarantee that a suitable number of points (candidate configurations) will be sampled from the parameters space, as well as their well distributed sampling.

Considering that one of the approaches used here for generating regression models is based on a multiple linear regression (as will be presented latter in this chapter), the suggested number of initial candidate configurations is based on a *rule-of-thumb* presented in [64], advising that is not recommendable to try estimate more than $n/3$ coefficients, where n is the number of observations. Thus, having a priori the number k of regressor coefficients to be estimated, it is recommended a set of initial candidates being not less than $n = 3k$. However, the method allows the user to enter with a different value.

Once the number of initial candidate configurations is defined, they are obtained by using either Latin Hypercube Sampling (LHS) [65, 66] or Sobol Sequence Sampling [67]. The LHS technique guarantees that if m parameter values for m candidate configurations have to be sampled in a domain $[\underline{v}_i, \overline{v}_i]$ of a parameter X_i, these values will be equally distributed among the m equal sub-ranges of the domain. The Sobol Sequence uses a base of two to create successively finer partitions of the unit interval and generate points equally distributed through these partitions. These sort of sampling result in a sampling that uniformly covers the parameters space.

Before proceeding, it is important to understand that performance degradation can occur if the parameters being tuned can assume values on possibly very different scales – e.g., in the case of polynomial mutation [68], the rate parameter exists in the $[0, 1]$ interval, while η can in principle assume any non-negative value. This is a well-known issue in the regression and machine learning literature [69], which can be avoided when tuning numerical parameters by simply re-scaling all parameters to a common scale, e.g., $[0, 1]$, as presented by the following equation:

$$\theta'_{i(l)} = \frac{\theta_{i(l)} - \theta_{(l,min)}}{\theta_{(l,max)} - \theta_{(l,min)}}, \quad i = 1, \ldots, m; \tag{3.2}$$

where $\theta_{i(l)}$ is the value of the l-th component of a candidate configuration θ_i, and $\theta_{(l,min)}$ and $\theta_{(l,max)}$ denote the lower and upper allowed values for the l-th parameter being tuned. Notice that this require all parameters to have upper and lower limits, which is generally not a problem – even for parameters that are theoretically unbounded, it is generally possible to define reasonable bounds based on theory or previous experience.

3.2.2 Evaluation of Candidate Configurations

At the beginning of each iteration the candidate configurations belonged to the *Elite Set* (best candidates) are evaluated on a set of instances in order to generate the response surfaces. Since the goal is to approximate the average performance of the algorithm on a set of instances, a crucial question that have to be answered is how many instances have to be used before building the response surfaces.

In [70], Pérez *et al.* analyze the parameters of the Irace tuning method. Among the parameters analyzed, one is the number of instances used before performing a statistical test (Friedman Test). Empirical tests allowed them to suggest five instances is the smallest amount that provides statistical evidence for beginning to perform the statistical tests at each iteration, with reduced risks of accidentally discarding good candidate configurations.

Although the proposed method uses different statistical tools in relation to Irace, it is applied to the same context and is expected to handle data with the same levels of variability. Therefore, it is suggested that the first regression models at the first iteration have to be generated after evaluating the candidate configurations on a minimum of 5 instances. At each new iteration, a number of q' new instances is added, the best candidates are evaluated on them, and a new set of regression models is generated.

Another important question when evaluating the candidate configurations on a set of instances relates to the number of repeated runs required for each configuration on each

instance. Although there is a widely adopted methodology of maximizing this quantity n as much as the computational budget allows, in order to obtain a more accurate measure of the candidate average performance on each instance, this allocation of resources has been proven to be suboptimal. A theoretical result demonstrated by Birattari ([71]) shows that: assuming a computational budget of N experiments (evaluations of candidate configurations on instances), the setting that yields the minimum *across-instances* variance is that for which the number of instances will be maximized, with each candidate configuration executed once on each instance.

As the objective of this parameter tuning method is to find the best candidate configuration for a class of problems, it is desirable a setting of experiments which minimizes the *across instances* variance. Therefore, initially, each candidate configuration is evaluated once on each instance. Another important issue is that: given the possibly heterogeneous nature of the tuning instances and the expected variations of performance of different configurations, it is possible that the distributions of the performance of candidate configurations on the instances exist on very different scales.

While some regression models, particularly quantile regression [72], can deal with these differences of scale relatively well, most have their performance heavily degraded in the presence of such large scale differences and heterogeneity of variances. To alleviate these particular problems, the performance of candidate configurations on the tuning instances is calculated by running the configurations on the test instances and transforming the output (i.e., the value of the quality indicator used) to a common scale.

In this sense, before being used for generating response surfaces, all values $p_{\theta_i\gamma_j}$, representing the performance of a candidate configuration i on an instance j, are linearly scaled to the interval [0, 1], according to:

$$p_{\theta_i\gamma_j} = \frac{p_{\theta_i\gamma_j} - p_{min(\gamma_j)}}{p_{max(\gamma_j)} - p_{min(\gamma_j)}} \tag{3.3}$$

where:

$p_{min(\gamma_j)}$: the smallest observed value of $p_{\theta_i\gamma_j}$ for an instance j, $\forall i = 1, \ldots, m$
$p_{max(\gamma_j)}$: the largest observed value of $p_{\theta_i\gamma_j}$ for an instance j, $\forall i = 1, \ldots, m$

Moreover, considering that throughout the process of tuning the number of instances on which each candidate configuration is evaluated is different from one to another (because at each iteration only a set of *Elite* candidates are maintained for evaluating on further instances), the overall performance of a candidate configuration p_{θ_i} on all instances q_k used at a $kth-$ iteration is calculated as:

$$p_{\theta_i} = S(p_{\theta_i}) * \frac{q_i}{q_k} \tag{3.4}$$

where:

q_i : the number of instances on which the candidate θ_i was evaluated
$S(p_{\theta_i})$: is a summarizing function (e.g.: mean, median, etc), considering all performance values $p_{\theta_i \gamma_j}$ of a configuration θ_i on all instances on which it was evaluated

Once the overall performance p_{θ_i} is calculated for all candidates using the equation 3.4, they are used with the data about candidate configurations scaled to the interval [0,1] to generate response surfaces by the regression modelling.

3.2.3 Generating response surfaces

The generation of response surfaces to be optimized at each iteration is performed in two steps: firstly, a regression model is fitted using a modeling technique of choice. In the current work, we investigate the use of multiple linear regression, more specifically 4 techniques: Ordinary Least Squares (OLS), Quantile, Lasso (least absolute shrinkage and selection operator) and Ridge regression; but other techniques could be used (like radial basis functions) for generating response surfaces [1]. Secondly, the model obtained is perturbed several times, generating new response surfaces. The details on how these steps occur are provided below.

3.2.3.1 Generating Linear Regression Models

As it is expected the algorithm behaviors to be described by multimodal functions, suitable linear regression models must be chosen for obtaining representatives response surfaces, that is, response surfaces that describes approximately the algorithm behavior. Since is difficult to know a priori the order of these models, the method allows the user to specify an initial order, and generate response surfaces of this order if there is enough performance data.

On the contrary, the method initiates generating models of second order and gradually increase this order as the performance data become enough, respecting the maximum order informed by the user. Thus, the order of regression models informed by the user can be seen as the maximum expected order of the models which represent the algorithm

[1] more details on regression modeling techniques are provided in Appendix A.

behavior. For obtaining response surfaces in a variety of experiments of the initial tests for the proposed method, a third order linear regression model is used, and it can be described by the equation:

$$y = \beta_0 + \sum_{i=1}^{m} \beta_i x_i + \sum_{i=1}^{m} \sum_{j \geq i}^{m} \beta_{ij} x_i x_j + \sum_{i=1}^{m} \sum_{j \geq i}^{m} \sum_{k \geq j}^{m} \beta_{ijk} x_i x_j x_k \tag{3.5}$$

Using the equation above it is possible to model the relationship between the parameter values and the average algorithm performance up to third-order effects. After fitting the regression coefficients β it is possible to obtain the standard error of each coefficient β_j, for each of the 4 regression techniques cited (OLS, Quantile, Lasso and Ridge).

Based on the maximum likelihood regression model obtained by using the multi linear regression, new regression models are generated. Let $\widehat{\mathbf{se}}_{\boldsymbol{\beta}}$ the vector of standard errors associated to the coefficients β of the first model. New regression models are generated by sampling values for their coefficients β' in the interval defined by $\boldsymbol{\beta} \pm \widehat{\mathbf{se}}_{\boldsymbol{\beta}}$, using an uniform probability distribution, that is, each new coefficient β'_k of a new model is defined as: $\beta'_k \sim \mathcal{U}\left(\beta_k - \widehat{se}_{\beta_k}, \beta_k + \widehat{se}_{\beta_k}\right),\ \forall k$.

For instance, let the third-order regression model $M_1 : Y = 3.52 + 5.74X_1^2 + 2.55X_1X_2 + 2.11X_2^3$ which associate the parameters values of X_1 and X_2 to the algorithm performance. Let $\widehat{\mathbf{se}}_{\boldsymbol{\beta}} = [1.4, 2.6, 1.2, 1.1]$ the vector of standard errors associated to the estimates of the intercept β_0 and the 3 regressor coefficients β_1, β_2 and β_3 of M_1. For a new perturbed model M_2, the estimates of the intercept β'_0 and the coefficients β'_1, β'_2, β'_3 would be calculated as: $\beta'_0 \sim \mathcal{U}\left(3.52 - 1.4, 3.52 + 1.4\right)$, $\beta'_1 \sim \mathcal{U}\left(5.74 - 2.6, 5.74 + 2.6\right)$, $\beta'_2 \sim \mathcal{U}\left(2.55 - 1.2, 2.55 + 1.2\right)$ and $\beta'_3 \sim \mathcal{U}\left(2.11 - 1.1, 2.11 + 1.1\right)$

As the regression techniques are based on mathematical formulations with different properties, the standard error se_{β_j} of each regression coefficient β_j is determined in a different way, depending on the technique used. For each regression technique, the standard errors are calculated as follows:

1. **OLS Regression**: When using OLS Regression, each standard error se_{β_j} of each regression coefficient j is defined as:

$$se_{\beta_j} = \sqrt{\hat{\sigma}^2 C_{jj}} \tag{3.6}$$

where C_{jj} is j-th diagonal element of $(\mathbf{X}'\mathbf{X})^{-1}$ (see section A.1), and $\hat{\sigma}^2$ is the estimate of the common variance of the residuals (which are assumed i.i.d. Gaussian in multiple linear regression).

2. **Quantile Regression**: The vector of standard errors $\widehat{\mathbf{se}}_{\boldsymbol{\beta}}$ is calculated using the option "ker" of the function *summary.rq* of the *R package "quantreg"* [73]. This option is used because it was the option that presented the best results of the standard errors, when testing *summary.rq* with simulated data (that is, data for which the actual value of each regression coefficient was known). Once the vector $\widehat{\mathbf{se}}_{\boldsymbol{\beta}}$ is determined, each coefficient β'_k of a new regression Model is calculated as: $\beta'_k \sim \mathcal{U}\left(\beta_k - \widehat{se}_{\beta_k}, \beta_k + \widehat{se}_{\beta_k}\right),\ \forall k$.

3. **Lasso and Ridge Regression**: The Ridge and Lasso regression Models were implemented using the *R package "hqreg"* ([74]), which does not provide an approach of calculating the vector $\widehat{\mathbf{se}}_{\boldsymbol{\beta}}$ of standard errors associated to the regression coefficients. Thus, it was proposed an approach based on a re-sampling method to determine the vector $\widehat{\mathbf{se}}_{\boldsymbol{\beta}}$ of a Lasso or Ridge model M.

   More specifically, firstly a Lasso/Ridge Model M is obtained using all set of n available algorithm performance data, and all coefficients that were shrunk down to zero are removed from this model. Secondly, n other models $\{M_1, \ldots, M_n\}$ are generated using a leave-one-out (LOO) strategy: the resulting polynomial from M is then used as a basis for fitting n new models, each of which is fitted using a different combination of $n-1$ algorithm performance data. Then, the vector $\widehat{\mathbf{se}}_{\boldsymbol{\beta}}$ of the coefficients β associated to the model M is calculated as the sample standard deviation of the coefficient values β from the n perturbed models $\{M_1, \ldots, M_n\}$. Once the vector $\widehat{\mathbf{se}}_{\boldsymbol{\beta}}$ is determined, each coefficient β'_k of a new regression model is calculated as showed for the previous types of regression modelling.

3.2.4 Model Optimization

The task of optimizing the set of response surfaces obtained has the goal of analyzing several likely variations of the average algorithm behavior (represented by the response surfaces) and finding the best parameter values (for minimization or maximization) associated with them.

Considering that the response surfaces are at least reasonable preliminary predictions of the true average behavior of the algorithm, optimizing them will yield a set of estimated best points (candidate configurations) which will be used jointly with the already known points for generating new regression models at the next iteration. It is expected that these new models will be biased towards good regions of the parameter space as the best points drive the generation of models that take into account regions of the parameter

space with good results. Moreover, as the iterations progress, the candidate configurations are evaluated on new instances and the inference about the average performance of the candidate configurations on the set of available instances becomes more accurate.

After a number of iterations, the pool of candidate configurations will be concentrated in one or more regions of the parameters space, that will ideally represent local optima regions, possibly including the global optimum region of the *algorithm configuration problem* (section 2.2).

The optimization methods used depends on the nature of the response surfaces generated. Exact methods that can handle the characteristics of functions generated can be used; or stochastic algorithms, which are attractive because their strategies for escaping local optima. In this work we experiment one approach of optimizing the response surfaces generated by the proposed parameter tuning method: a version of the Nelder-Mead Algorithm, a direct search optimization method.

One circular reasoning question may arise over the use of algorithms for the task of optimizing models: "How to tune the algorithm used in a step of a method that has the objective of tuning algorithms?" The answer for this question passes by the notion that it is not necessary to tune this algorithm. Even with standard parameter values, it can usually achieve reasonable results, and its performance will be improved along the iterations, as the candidate configurations will concentrate in some regions of the parameters space and the models to be optimized will be biased to narrower regions of that space.

3.2.5 MetaTuner Algorithm

The general aspects of the proposed framework can be easily explained from the structure presented in Algorithm 8 [2]. The method starts by sampling a few configurations, which are evaluated on a randomly sampled initial set of tuning instances (lines 1-5). The performance results obtained are then used for fitting a regression model of the expected performance of configurations on the problem class of interest (lines 7-17). The regression model is then subject to perturbations (e.g., by perturbing the fitted parameters), resulting in a number of additional response surfaces. For each surface (including the unperturbed one) an optimization process is executed, returning a new candidate configuration which maximizes the value of the estimated average performance value for that model (lines 18-22).

[2] An open-source implementation is available in the form of R package *MetaTuner* (`https://github.com/fcampelo/MetaTuner`). Details about the structure of the tool, as well as a full usage example, are available in the package documentation.

Algorithm 8 Proposed tuning framework

Require: Search space (Ω); Tuning instances (Γ_S); number of initial configs. (m_0); number of new configs/iter. ($m_\star$); number of initial instances (N_0); number of addit. instances/iter. ($N_\star$); size of archive ($n_\mathcal{E}$).

1: $t \leftarrow 0$
2: $\mathcal{A}^{(t)} \leftarrow$ GenerateInitialSample (Ω, m_0) ▷ Sample initial configurations
3: $\Gamma_\mathcal{A}^{(t)} \leftarrow$ SampleWithoutReplacement $(\Gamma_S, N_0 - N_\star)$ ▷ Sample initial instances
4: $\mathcal{P}_\mathcal{A}^{(t)} \leftarrow$ EvaluateConfigurations $\left(\mathcal{A}^{(t)}, \Gamma_A^{(t)}\right)$ ▷ Evaluate configs on instances
5: $\mathcal{E}^{(t)} \leftarrow \mathcal{A}^{(t)}$ ▷ Initialise elite archive
6: **while** Stop criteria not met **do**
7: $t \leftarrow t + 1$
8: **if** New instances available **then**
9: $\Gamma' \leftarrow$ SampleWithoutReplacement $\left(\Gamma_S \backslash \Gamma_\mathcal{A}^{(t-1)}, N_\star\right)$ ▷ Sample new instances
10: $\mathcal{P}' \leftarrow$ EvaluateConfigurations $\left(\mathcal{E}^{(t)}, \Gamma'\right)$ ▷ Eval. elite configs on new instances
11: $\Gamma_\mathcal{A}^{(t)} \leftarrow \Gamma_\mathcal{A}^{(t-1)} \bigcup \Gamma'$ ▷ Update archive of instances visited
12: $\mathcal{P}_\mathcal{A}^{(t)} \leftarrow$ Update $\left(\mathcal{P}_\mathcal{A}^{(t-1)}, \mathcal{P}'\right)$ ▷ Update archive of config. performances
13: **else**
14: $\Gamma_\mathcal{A}^{(t)} \leftarrow \Gamma_\mathcal{A}^{(t-1)}$
15: $\mathcal{P}_\mathcal{A}^{(t)} \leftarrow \mathcal{P}_\mathcal{A}^{(t-1)}$
16: **end if**
17: $\mathcal{S}_1^{(t)} \leftarrow$ FitModel$\left(\mathcal{A}^{(t-1)}, \mathcal{P}_\mathcal{A}^{(t)}\right)$ ▷ Fit regression model
18: $\theta_1'^{(t)} \leftarrow$ Optimise $\left(S_1^{(t)}\right)$ ▷ Find configuration that optimises $\mathcal{S}_1^{(t)}$
19: **for** $j \in \{2, \ldots, m_\star\}$ **do**
20: $\mathcal{S}_j^{(t)} \leftarrow$ PerturbModel $\left(S_1^{(t)}\right)$ ▷ Generate perturbed model
21: $\theta_j'^{(t)} \leftarrow$ Optimise $\left(S_j^{(t)}\right)$ ▷ Find configuration that optimises $\mathcal{S}_j^{(t)}$
22: **end for**
23: $\mathcal{C}^{(t)} \leftarrow \left\{\theta_j'^{(t)} : j = 1, \ldots, m_\star\right\}$
24: $\mathcal{P}_\mathcal{C}^{(t)} \leftarrow$ EvaluateConfigurations $\left(\mathcal{C}^{(t)}, \Gamma_\mathcal{A}^{(t)}\right)$ ▷ Evaluate candidate configs
25: $\mathcal{A}^{(t)} \leftarrow \mathcal{A}^{(t-1)} \bigcup \mathcal{C}^{(t)}$ ▷ Add candidate configs to archive
26: $\mathcal{P}_\mathcal{A}^{(t)} \leftarrow$ Update $\left(\mathcal{P}_\mathcal{A}^{(t)}, \mathcal{P}_\mathcal{C}^{(t)}\right)$ ▷ Update archive of config performances
27: $\mathcal{E}^{(t)} \leftarrow$ SelectKBest $\left(\mathcal{A}^{(t)}, \mathcal{P}_\mathcal{A}^{(t)}, K = n_\mathcal{E}\right)$ ▷ Update elite archive
28: **end while**
29: $\mathcal{P}_\mathcal{E}^{(t)} \leftarrow$ RetrieveElitePerformances$\left(\mathcal{E}^{(t)}, P_\mathcal{A}^{(t)}\right)$
30: **return** Elite configurations ($\mathcal{E}^{(t)}$) and their estimated performance ($\mathcal{P}_\mathcal{E}^{(t)}$).

These new candidate configurations are then evaluated on all instances sampled so far, and added to an archive. Finally, the archive is truncated to a given size, maintaining only the candidate configurations with the best expected performance value for the problem class (lines 23-27). The whole process then iterates by sampling a few more instances (if available), and proceeds until a predefined stopping condition is reached. If no new instances are available, the process simply continues generating new candidate configurations at each iteration, which are then evaluated on all instances. This process proceeds until a predefined stopping condition is reached.

Despite the specific implementation details of the main steps of the algorithm described

above (Generation of initial candidates, Evaluation of Candidates, Generation of regression Models and Model Optimization), it is important to point out that the main ideas behind the method can be adapted to a wider variety of contexts by varying the operation of these steps. The specific implementation presented here, is just one possible instantiation of the main ideas of the framework.

Other types of sampling could be used for generating the initial candidates, for instance using some specific probability distribution. The step of evaluation could be changed, maybe considering the sum of rankings instead of the average normalized performance of the candidates, or average rankings, trimmed mean, median, etc. Other approaches of obtaining prediction models (here represented by response surfaces) could be used, like higher order linear regression models, several models of radial basis functions, regression trees, etc. Finally, for optimizing the prediction models, a wide sort of algorithms are available. The method in its details is as important as the main ideas of the framework, which allows several versions of this method to be developed for specific applications.

3.2.6 MetaTuner in the context of parameter tuning methods

It should be clear by now that the proposed method is situated within the scope of SMBO methods for parameter tuning. We finish this section by highlighting the similarities and differences between our work and other methods in the literature.

Although the results of the works mentioned in the section 2.4 are useful to provide a comparative viewpoint of the cited parameter tuning methods, it is interesting to have an overview of them, considering a qualitative perspective. The table 3.1 classifies the most known parameter tuning methods considering three aspects: if the tuning scenario is related to one instance (single-instance) or several instances (multi-instances), the type of parameters they can tune (only numerical or all sort of parameters); and about the output information, that is, if they provide to the user only the information about the best configuration or the best configuration and a model which allows the user to understand more about the algorithm behavior and the relevance of parameters. Observing these aspects the user can decide which parameter tuning method is more suitable to his/her tuning scenario.

According to the table 3.1, considering the types of parameters the algorithms can have, the most suitable methods are Irace, ParamILS, SMAC and CRS-Tuning, since they can deal with numerical and categorical parameters. Thus, they can work with a wider spectrum of tuning scenarios. An example are when tuning several numerical and categorical parameters of solvers, which usually have many mixed parameters. Another important issue which arises during the tuning process (sometimes can be a crucial issue)

TABLE 3.1: Qualitative classification of parameter tuning methods.

Method	**# Instances**		**Parameters**		**Output Information**	
	1	*	Numerical	All types	θ^*	θ^* and model/relevance
Irace		X		X	X	
ParamILS		X		X	X	
SMAC		X		X		X
SPO	X		X			X
REVAC	X		X			X
Bonesa		X	X		X	
CRS-Tuning		X		X	X	
MetaTuner		X	X			X

is the capability of learning about the algorithm behavior. The methods SMAC, SPO and REVAC can theoretically provide the user models that relate the parameters and the algorithm performance, which is certainly better than inform only what are the best parameter values.

At last but not least, the tuning process can be focused on to find out the best parameter values when running an algorithm on one instance or problem (single-instance) or on a set of instances or problems (multi-instances). Considering the single-instance scenario, SPO and REVAC were designed specially for it, although in [59] Montero et al. showed that REVAC can be easily adapted for multi-instances. SPO, for its turn, specifically drives the computational budget to be used for evaluating more times those best candidates on the instance used. It is important to highlight here that Bonesa provide a set of best candidate configurations, based on a pareto-strenght scheme, in which the performance of candidate configurations on each instance used is considered one objective. This information allows the user to contrast the several non dominant candidates and to balance the pros and cons of each one.

The parameter tuning method presented in this work can be classified as: multi-instance, to only numerical parameters, although its modular structure can be easily adapted to work with mixed parameter spaces, by modifying the regression model (e.g., using generalized linear models with mixed inputs [75]) and optimization approaches (e.g., using mixed-variables or bilevel optimization methods) [3]; and that provides an algorithm model behavior and information relevance. In this case, the method that cover all these characteristics is SMAC; with REVAC being another that provides information about parameters relevance.

In relation to SMAC, the current implementation has the disadvantage of not yet dealing with categorical parameters. It does, however, present certain advantages that justify its

[3] Even though the particular instantiation presented in this work deals only with numeric parameters, the framework is designed to allow extensions to categorical ones as well.

proposal. First, it uses robust regression models such as ridge and lasso regression, which works well under heteroscedasticity and in the presence of outliers (which can arise, e.g., from heavily heterogeneous problem classes), and they incorporate implicit attribute selection, which also allows the method to focus on finding out the most important parameters, as well as to return regression models even when the number of parameters is large. In relation to REVAC our method is capable to return regression models that provide information about the main effects of the parameters, as well as the relevance of the interaction effects of the parameters, whereas REVAC can deal with just the main effects of the parameters.

Also, the explicit consideration of modelling uncertainty - which motivates the use of perturbed models in the search phase of the method - allows a more comprehensive search, and can provide additional evidence at the end of the tuning process of the quality of the models fit and, consequently, their expected explanatory and predictive abilities for the performance of the tuned configurations when solving new instances from the same problem class.

Finally, even though the proposed framework does not necessarily employ evolutionary algorithms in the optimization phase, it bears some similarities to other methods belonging to the wider class of Surrogate-Assisted Evolutionary Algorithms (SAEAs) [76–79], to which several SMBO methods also belong. SAEAs are commonly employed in the solution of optimization problems in which the computational cost of evaluating the objective or constraint functions is particularly high, which happens often in applied contexts [77, 80–82]. Actually, it is possible to express the parameter tuning problem as a noisy, expensive optimization problem: the formulation presented in 2.2 already suggests tuning as an optimization problem, where the objective is the maximization of the expected performance of the method to a problem class of interest.

The “expensive” part comes from the fact that the performance evaluation of a given candidate configuration requires the execution of not only one, but several runs of the algorithm equipped with that configuration on different instances of the problem class of interest. The “noisy” part comes from the uncertainties in the evaluation of the expected performance, which is estimated based on a finite number of runs executed on a finite subset of problem instances. This view of the tuning problem is used to motivate a simulation model employed in the experiments described in Section 4.2.1, where we investigate the performance of the proposed tuning method.

Chapter 4

Experimental Results

4.1 Introduction

To illustrate the use of the proposed approach we performed three experiments, in which we analyze the abilities and weakness of our framework, besides comparing them with some well known parameter tuning methods in the metaheuristics literature: Irace, ParamILS and SMAC. Some other remainder parameter tuning methods from the literature mentioned in this work were not considered in this experiment for some reasons: SPO is single instance, and these experiments represent multi-instances scenarios; Bonesa returns an information (a pareto set of configurations) that is innapropriated to compare against those returned by MetaTuner, Irace, ParamILS and SMAC; and no available standard implementation of REVAC was found.

In the first experiment, a simulation model was used to describe a hypothetical average performance surface, over which random noise was added to simulate the across-instances and within-instance variance commonly experienced in real tuning scenarios. The objective of this first experiment is to evaluate the behaviour of the proposed tuning method under known, controllable conditions, which allows an exploration of the abilities and limitations of MetaTuner.

The second and third experiments contrast the performance of the proposed method against the three others already mentioned parameter tuning methods from the literature, in real-valued parameter tuning problems. In the second experiment three parameters of a well-known single objective algorithm are tuned, using two sets of challenging problems. Four different instantiations of MetaTuner are tested, in order to investigate the strengths and weaknesses of different variations of the proposed approach in relation to existing methods.

The third experiment is a comparison between four distinct variations of MetaTuner against four other tuning approaches: Irace, ParamILS and SMAC, as well as a simple random sampling method, in a scenario with a very limited number of available tuning instances. The analysis of relevance of the parameters returned by the regression models provided by the MetaTuner versions is performed in all experiments.

4.2 Experiments using a Simulation Model

4.2.1 The Simulation Model

In this experiment,[1] the expected performance of a (hypothetical) algorithm on a (also hypothetical) set of instances was represented by a simulation model with the following structure:

$$p^k_{\theta_i;\gamma_j} = p_{\theta_i;\Gamma} + \tau_{\theta_i;\gamma_j} + \epsilon^k_{\theta_i;\gamma_j} \tag{4.1}$$

where:

- $p^k_{\theta_i;\gamma_j}$ represents the performance value obtained by a configuration θ_i on an instance γ_j at the k-th run;
- $p_{\theta_i;\Gamma}$ is the expected value for the performance of the configuration θ_i on the whole instance class Γ, that is, the *grand mean* of the performance of that particular configuration for the problem class of interest. This value is defined using a function over the space of parameters Θ.
- $\tau_{\theta_i;\gamma_j}$ is the deviation between the expected performance value of θ_i on γ_j and the *grand mean* of θ_i on the instance class Γ. In this model the variance of the $\tau_{\theta_i;\gamma_j}$ values is used to simulate the *across-instance variance*;
- $\epsilon^k_{\theta_i;\gamma_j}$ is the deviation between the observed performance value of θ_i on γ_j at the k-th run and the expected performance of θ_i on γ_j. The variance of the $\epsilon^k_{\theta_i;\gamma_j}$ values is used in this model to simulate the *within-instance variance*;

Equation 4.1 models the performance $p^k_{\theta_i;\gamma_j}$ of a candidate configuration θ_i on an instance γ_j at the k-th run as an additive effects model composed of the grand mean $p_{\theta_i;\Gamma}$, the effect $\tau_{\theta_i;\gamma_j}$of instance γ_j on that grand mean, and the variability between runs of the configuration on that particular instance, $\epsilon^k_{\theta_i;\gamma_j}$.

[1] This experiment was performed in a Intel Core 2 Quad Machine, with 4 2.83GHz cores, 3.7 Gib of memory running Ubuntu 18.04.

For this experiment the effects $\tau_{\theta_i;\gamma_j}$ and $\epsilon^k_{\theta_i;\gamma_j}$ were generated using shifted exponential distributions with zero mean and variances σ^2_{AI} (for $\tau_{\theta_i;\gamma_j}$) and σ^2_{WI} (for $\epsilon^k_{\theta_i;\gamma_j}$). The grand mean $p_{\theta_i;\Gamma}$ was represented using analytic functions with known optima, to allow the assessment of MetaTuner as a tuning approach using a hypothetical algorithm dealing with known performance landscapes.

Two functions were used to model hypothetical performance landscapes:

- **Quadratic function**:

$$f(\boldsymbol{\theta}) = 2 + 100\theta_1^2 + 5\sum_{i=2}^{n}\theta_i \text{ , with } \begin{cases}\theta_1 \in [-10, 0] \\ \theta_i \in [0, 1], i = 2, \ldots, n\end{cases} \tag{4.2}$$

- **Ackley function**:

$$f(\boldsymbol{\theta}) = -a\ exp\left(-b\sqrt{\frac{1}{3}\sum_{i=1}^{3}\theta_i}\right) - exp\left(\frac{1}{3}\sum_{i=1}^{3}cos(c\theta_i)\right) + a + exp(1) \tag{4.3}$$

  with $\theta_i \in [-32.8, 32.8],\ i = 1, \ldots, n; \quad a = 20;\ b = 0.2;\ c = 2\pi$

The dimensions of the functions were varied between 2 and 8. The first function represents a smooth, "well behaved", response landscape, and all parameter tuning methods were expected to converge to the vicinity of the global optimum value ($\boldsymbol{\theta}^* = \mathbf{0}$). Moreover, the first parameter (θ_1) is much more influential than the others, a fact that should be captured by the regression models used in MetaTuner.

The second function is used to investigate the ability of the parameter tuning methods to explore performance landscapes with multiple local optima. This function is commonly used in simulated experiments with metaheuristics, and was chosen here to represent a much more challenging average performance landscape. Actual average performance surfaces are probably located in between the two extremes represented by these test models, which are used here to investigate general performance trends.

For each function and each dimension, all parameter tuning methods were executed 30 times, under a budget of $300n$ configurations to be evaluated.[2] The variances in the model were arbitrarily chosen as $\sigma^2_{AI} = 4$ and $\sigma^2 WI = 1$. MetaTuner was set up with an initial sampling of $m_0 = 20$ configurations generated by Latin Hypercube Sampling, $n_0 = 5$ initial instances randomly drawn from the tuning set, $m_i = 5$ new configurations generated at each iteration, and $n_i = 1$ new tuning instances added to the pool at each iteration. The median was used during the tuning process as the summary function

[2] This budget was determined by exploratory tests with similar functions, using Irace

for calculating the expected performance of each configuration in all methods, except SMAC. [3] Other initial parameters of SMAC, Irace and ParamILS were set as their standard configurations. Four versions of MetaTuner were used: using Linear (OLS), Quantile, Lasso or Ridge regression, all with a polynomial model of order 3 and using Nelder-Mead for optimizing the regression models.

For ParamILS the numerical parameters must be informed as a sequence of discrete values. For the Quadratic function, the possible values for θ_1 were $\{-10.0, -9.5, \ldots, -0.5, 0.0\}$, with initial value of -5.5; for all other parameters the possible values were determined as being equal to: $\{0.00, 0.05, \ldots, 0.95, 1.00\}$, with initial value 0.50. In the case of Ackley function, for all parameters the possible states were set as 21 equally-spaced values in the range ± 32.80, with an initial value of 0.5. This characteristic of ParamILS can represent and advantage to it, once that the best parameter values may be included in the set of discrete values, allowing the method to reach exactly the best parameter values. Therefore, if the best parameter values are not included in this initial information, ParamILS can not reach them.

4.2.2 Comparing the Parameter Tuning Methods

To compare the output of MetaTuner versions and other parameter tuning methods, the gap between the function values associated with the best candidate configurations achieved by each approach and the optimum value were analyzed. The Figure 4.1 illustrates the mean performances regarding the optimality gap.

Considering the quadratic function, Irace had clearly the worst performance among all methods, while Metatuner using Linear and Quantile models presented the best results. This was expected for Metatuner, as this simulated performance landscape can be easily represented by the polynomial form of the regression methods tested. The differences were statistically significant at the 95% confidence level (Friedman test, $p = 9.7 \times 10^{-7}$), with the Quantile version of Metatuner being significantly better (in terms of across-dimensions mean performance) to all other methods except Linear version.

As for the Ackley landscape, ParamILS exhibited a better overall performance, followed by Irace and SMAC. The Metatuner versions tested were not capable of adequately learning this more complex landscape, as it cannot be properly described by the regression models used (it may be improved by increasing the order of the regression models used). The differences were also detected as statistically significant ($p = 2.8 \times 10^{-9}$), with ParamILS performing significantly better than all other methods.

[3] The version of SMAC used in the experiments, available from https://www.cs.ubc.ca/labs/beta/Projects/SMAC/v2.10.02/quickstart.html , does not use the median.

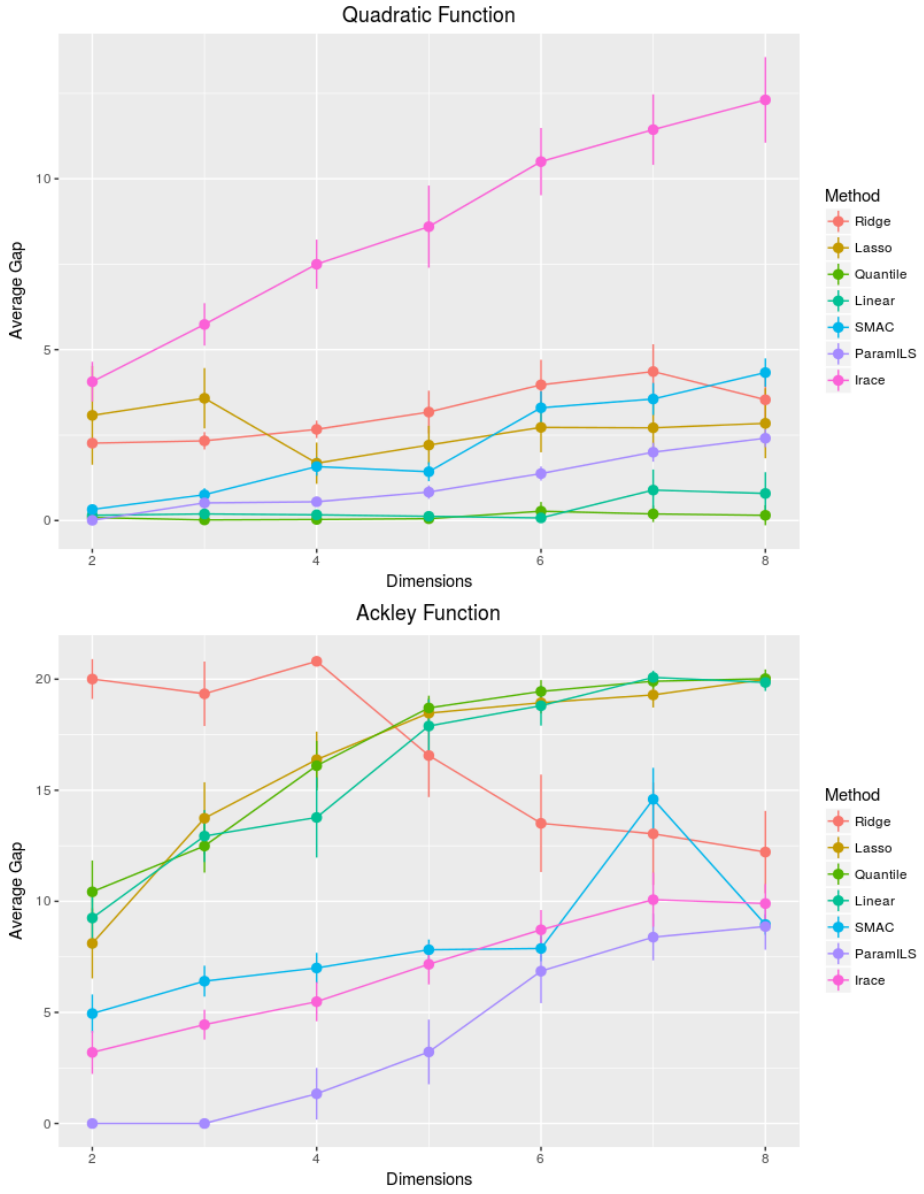

FIGURE 4.1: Point estimates and 95% confidence intervals of the mean optimality gaps, for the quadratic (top) and Ackley (bottom) simulated performance landscapes. MetaTuner versions were labelled based on the model employed.

In terms of runtime, the methods that rely on explicit modelling of the performance landscape (i.e., Metatuner and SMAC) resulted in runtimes that were not only substantially higher than those which do not (i.e., Irace and ParamILS), a problem that is amplified as the dimensions are increased, as illustrated in Figure 4.2.[4] This is an important point when tuning algorithms repeatedly, or for computationally "cheap" problem classes, but it can be argued that it is not a major obstacle against the use of model-based approaches for two main reasons: first, tuning is most often a one-time task, so the time required for this activity is generally not as important as the expected performance improvement

[4]The results for the quadratic performance landscape were similar.

it generates. Second, when tuning algorithm parameters for computationally expensive problems, even the considerably higher computational burden due to model-building can often be disregarded.

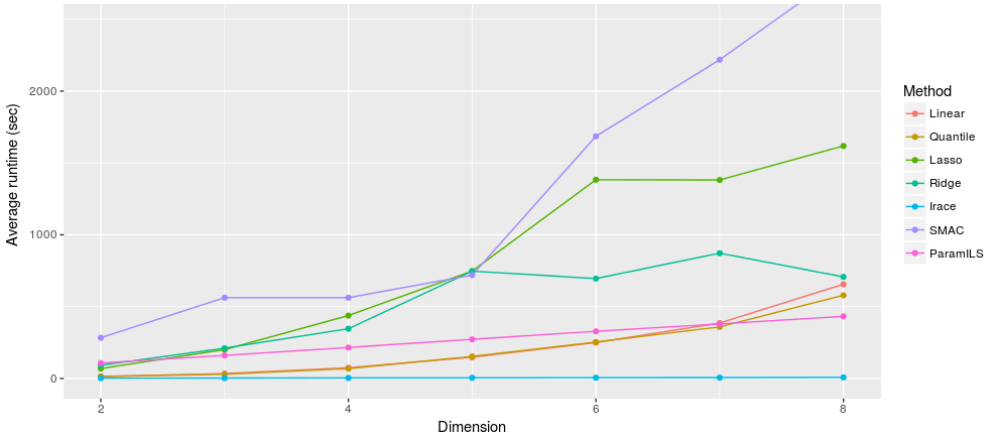

FIGURE 4.2: Average runtime of different tuning approaches for the Ackley performance landscape.

Finally, the use of simulated performance landscapes in this experiment provided information to evaluate the ability of Metatuner to detect the parameters that are the main drivers of performance, which, since Metatuner works on normalized parameter spaces, is simply a matter of examining which model components present the largest magnitudes (surprisingly, the version of SMAC used does not provide to the user a parameters model in its output, according to its user guide).

Tables 4.1 and 4.2 show a summary of the most relevant parameters, considering each version of MetaTuner and each dimension of Quadratic and Ackley Functions. Having both the algorithm performance and the parameter values scaled in the interval [0,1] when building the regression models, the most relevant parameters for each run were those with highest absolute values in the regression models.

The columns of these tables indicate which model parameters were interpreted by MetaTuner as "the most relevant" most often (**Freq#1**), second most-often (**Freq#2**) and third most-often (**Freq#3**). The two numbers in brackets represent how many times the parameter was the most relevant, followed by how many times it was among the 3 most relevant ones.

For instance, considering the Linear version of MetaTuner and the Quadratic Function of dimension 2, the results were the following: θ_1 was detected as the most relevant parameter in 21 runs, and among the 3 most relevant in 22 runs; θ_1^2 was detected as the most relevant parameter in 5 runs and among the 3 most relevant in 24 runs; and the interaction $\theta_1\theta_2^2$ was detected as the most relevant in 3 runs and among the 3 most relevant in 7 runs.

TABLE 4.1: More relevant parameters - Quadratic Function

Dimension	Version	Freq #1	Freq #2	Freq #3
2	Linear	θ_1[21;22]	θ_1^2[5;24]	$\theta_1\theta_2^2$[3;7]
	Quantile	θ_1[30;30]	-	-
	Lasso	θ_1[29;30]	θ_1^3[1;29]	-
	Ridge	θ_1[30;30]	-	-
3	Linear	θ_1[26;29]	$\theta_1\theta_2$[2;10]	θ_1^2[1;25]
	Quantile	θ_1[17;23]	θ_1^2[4;20]	$\theta_1\theta_2$[4;10]
	Lasso	θ_1[30;30]	-	-
	Ridge	θ_1[30;30]	-	-
4	Linear	θ_1[27;27]	θ_1^2[2;26]	$\theta_1\theta_3$[1;7]
	Quantile	θ_1[29;30]	θ_1^2[1;30]	-
	Lasso	θ_1[30;30]	-	-
	Ridge	θ_1[30;30]	-	-
5	Linear	θ_1[30;30]	-	-
	Quantile	θ_1[30;30]	-	-
	Lasso	θ_1[29;30]	$\theta_3^2\theta_4$[1;1]	-
	Ridge	θ_1[30;30]	-	-
6	Linear	θ_1[30;30]	-	-
	Quantile	θ_1[30;30]	-	-
	Lasso	θ_1[30;30]	-	-
	Ridge	θ_1[30;30]	-	-
7	Linear	θ_1[17;22]	$\theta_1\theta_5$[2;4]	$\theta_1\theta_7$[2;4]
	Quantile	θ_1[9;14]	θ_1^2[3;15]	$\theta_1\theta_5$[3;8]
	Lasso	θ_1[30;30]	-	-
	Ridge	θ_1[30;30]	-	-
8	Linear	θ_1[29;29]	θ_7[1;5]	-
	Quantile	θ_1[30;30]	-	-
	Lasso	θ_1[30;30]	-	-
	Ridge	θ_1[30;30]	-	-

For the quadratic case, all versions of Metatuner were able to detect θ_1 as the most relevant parameter with fairly high consistency, i.e., in near all runs. It can be observerd from the table 4.1 that the versions Ridge and Lasso were more effective for this task. Furthermore, it is worth highlighting that the main objective in this issue is to identify which parameter is more relevant (in this case, θ_1), and no necessarily its actual form of occurrence (in this case, θ_1^2).

Unlike the quadratic scenarios, the models output by Metatuner for the Ackley problems were more heterogeneous in terms of which parameters had the largest coefficients. This was expected, as the Ackley function does not have any parameter that stands out in terms of its influence on the function value - in a sense, the symmetry of this function means that all parameters have essentially the same relevance.

TABLE 4.2: More relevant parameters - Ackley Function

Dimension	Version	Freq #1	Freq #2	Freq #3
2	Linear	$\theta_1\theta_2$[11;16]	θ_2^2[7;16]	θ_1^2[6;12]
	Quantile	$\theta_1\theta_2$[12;17]	θ_2^2[7;18]	θ_1^2[7;12]
	Lasso	θ_1^3[12;21]	θ_2^3[8;20]	$\theta_1\theta_2$[4;9]
	Ridge	θ_2[30;30]	-	-
3	Linear	$\theta_1\theta_3$[6;11]	$\theta_2\theta_3$[5;9]	θ_1^2[5;6]
	Quantile	$\theta_1\theta_2$[10;13]	θ_3^2[6;11]	θ_2^2[5;10]
	Lasso	θ_2^3[3;11]	θ_2[3;8]	θ_3[2;7]
	Ridge	$\theta_2\theta_3$[23;25]	$\theta_2^2\theta_3$[3;3]	θ_3[2;24]
4	Linear	θ_1^2[6;12]	θ_3^2[4;9]	θ_4^2[4;9]
	Quantile	θ_2^2[8;10]	θ_1^2[6;11]	θ_3^2[4;10]
	Lasso	θ_3^3[3;6]	$\theta_1\theta_2\theta_3$[3;4]	$\theta_1\theta_3^2$[3;3]
	Ridge	$\theta_2\theta_3$[11;18]	$\theta_3\theta_4$[6;13]	θ_4[5;19]
5	Linear	θ_3^2[6;11]	θ_4^2[5;8]	θ_1^2[2;8]
	Quantile	θ_4^2[6;11]	θ_3^2[4;8]	θ_2^2[3;14]
	Lasso	θ_3^3[3;3]	θ_4^3[2;2]	$\theta_1\theta_3\theta_5$[1;2]
	Ridge	θ_5[14;24]	θ_3[3;16]	$\theta_3\theta_4\theta_5$[3;3]
6	Linear	θ_6^2[5;9]	$\theta_1\theta_2$[4;8]	θ_2^2[4;7]
	Quantile	θ_5^2[6;8]	θ_4^2[5;9]	θ_6^2[4;6]
	Lasso	θ_5[2;2]	$\theta_2\theta_4\theta_6$[1;3]	θ_2[1;2]
	Ridge	$\theta_2\theta_3\theta_4$[8;11]	θ_2^3[4;8]	θ_3[3;6]
7	Linear	θ_6^2[2;6]	θ_4^2[2;6]	θ_7^2[2;4]
	Quantile	θ_6^2[5;7]	θ_5^2[5;7]	θ_2^2[4;5]
	Lasso	$\theta_1\theta_2\theta_7$[1;2]	$\theta_4\theta_6^2$[1;1]	$\theta_4\theta_5\theta_7$[1;1]
	Ridge	θ_2^2[3,5]	θ_5^2[3;5]	θ_3[2;5]
8	Linear	θ_4[6;10]	θ_2[5;8]	θ_5[4;10]
	Quantile	θ_5[5;9]	θ_7[4;7]	θ_4[4;6]
	Lasso	$\theta_3\theta_6$[2;3]	$\theta_5\theta_8$[2;2]	$\theta_3\theta_5$[1;4]
	Ridge	θ_5^2[25;30]	θ_2^2[3;19]	θ_3^2[1;15]

While preliminary, the results of this experiment using the simulation model can yield some interesting insights: first, they suggest that MetaTuner may be able to discover the underlying structure of the response surface in cases where the structural form of the regression models used is adequate, despite the presence of noise due to within-instances and between-instances variances – e.g., in the case of the quadratic model, where the general shape of the performance surface can be described by the polynomial form used in MetaTuner.

4.3 Tuning DE Parameters

In this experiment, the parameter tuning methods were used for tuning parameters of a "standard" Differential Evolution, DE/rand/1/bin[5]. Three parameters were selected for tuning: the mutation factor $F \in [0.1, 5]$, the crossover rate CR $\in [0, 1]$, and the multiplier $K \in [10, 20]$, used to calculate the population size as $N_{pop} = K \times dim$, with dim the dimension of the problem being solved. The stop criterion of the DE was the use of $10000 \times dim$ objective function evaluations.

The tuning process was analyzed for two optimization scenarios: a first one consisting of similar problem instances, and a second with more heterogeneous problems. In the homogeneous scenario, the DE was used to solve 20 optimization problems sampled from functions 15 and 21 of the BBOB benchmark set [84] with dimensions $2, 4, 6, \ldots, 40$. Four versions of MetaTuner used in the prior experiment, as well as SMAC, Irace and ParamILS were considered. Each method was run 30 times, and at each run a budget of 1500 algorithm evaluations was used.[6] Each of the 30 best configurations returned by each method was then used to solve 19 *validation instances*, sampled from the same BBOB functions but with dimensions $3, 5, 7, \ldots, 39$. Each best configuration was run 30 times on each validation instance, and its performance considered as being the average of the 19 means related to the validation instances. The overall performance of each parameter tuning method was represented by the performance distribution data related to the 30 best configurations returned.

For the heterogeneous scenario the training set was formed by 30 functions sampled from functions BBOB 1 to 24 with dimensions between 8 and 11. The validation set was composed of 20 other functions sampled from the same set. The same budget of the former scenario was used for each tuning run. With the exception of the tuning budget, all parameters of MetaTuner, SMAC, Irace and ParamILS were set as in the prior experiment.[7] Considering ParamILS, the possible values of each parameter were the following: $F \in [0.1, 0.35, 0.60, \ldots, 5]$ with initial value equal to 2.35; $CR \in [0, 0.05, 0.10, \ldots, 1]$ with initial value equal to 0.5; and $K \in [10, 11, 12, \ldots, 20]$ with initial value of 15.

Figure 4.3 presents the distribution of the overall performance of the 30 best candidates returned by all methods for the homogeneous scenario. This figure suggests that Irace has the worst results, ParamILS and SMAC the best, and the MetaTuner versions were between these bounds but the distribution of observations presents a substantial overlap.

[5] The implementation available in the *R package ExpDE* [83] was used

[6] This budget was based on the work of Nannen and Eiben [26], which used a budget of 500 runs per parameter for an evolutionary algorithm

[7] This experiment was run in a Intel Xeon Silver machine, with 2.10GHz x 32 cores, 62.9 Gib RAM, running Ubuntu 19.04.

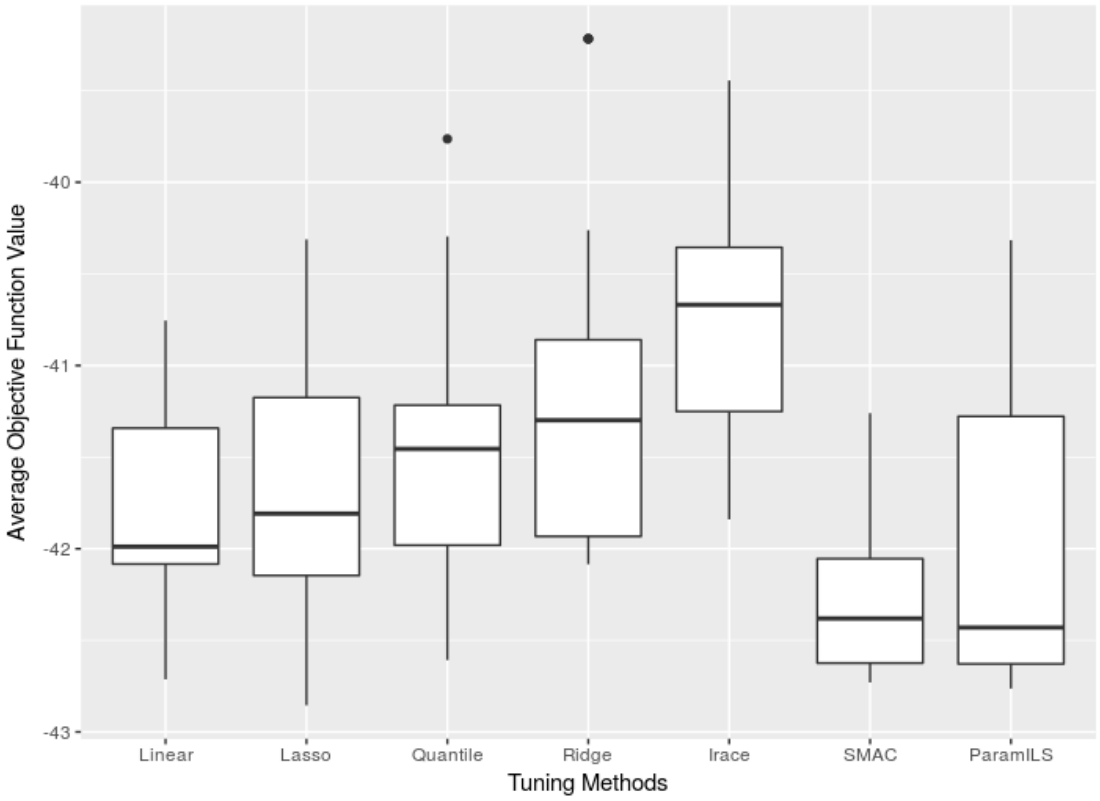

FIGURE 4.3: Overall mean performance - Homogeneous scenario. MetaTuner versions are indicated by the regression modelling.

To objectively evaluate these differences, an inferential approach was employed. Preliminary analyses suggested that the normality assumption could not be assumed, so a Kruskal-Wallis test [63] was performed to detect differences in this scenario, suggesting statistically significant differences at the 95% confidence level ($p < 3.65 \times 10^{-14}$). Pairwise Wilcoxon-Mann-Whitney tests were then performed to pinpoint the differences, indicating SMAC and ParamILS tied in first place; Linear and Lasso similar to each other in the second place; Quantile worse than Linear and similar to Lasso and Ridge; and Ridge better than Irace (significantly the worst).

Figure 4.4 presents the results observed for the heterogeneous scenario. This figure suggests that several methods present somewhat similar performances, with ParamILS presenting a somewhat less stable behaviour. The Kruskall-Wallis test indicated that at least some of the differences observed were statistically significant ($p = 1.53 \times 10^{-10}$), and subsequent Wilcoxon-Mann-Whithney tests detected SMAC and Irace in the first place, followed by ParamILS and Ridge (similar to each other); Lasso (similar to ParamILS and worse than Ridge). The versions Linear and Quantile of Metatuner presented the worst performance, with large outliers (due to this reason their performance data are not showed in the figure 4.4).

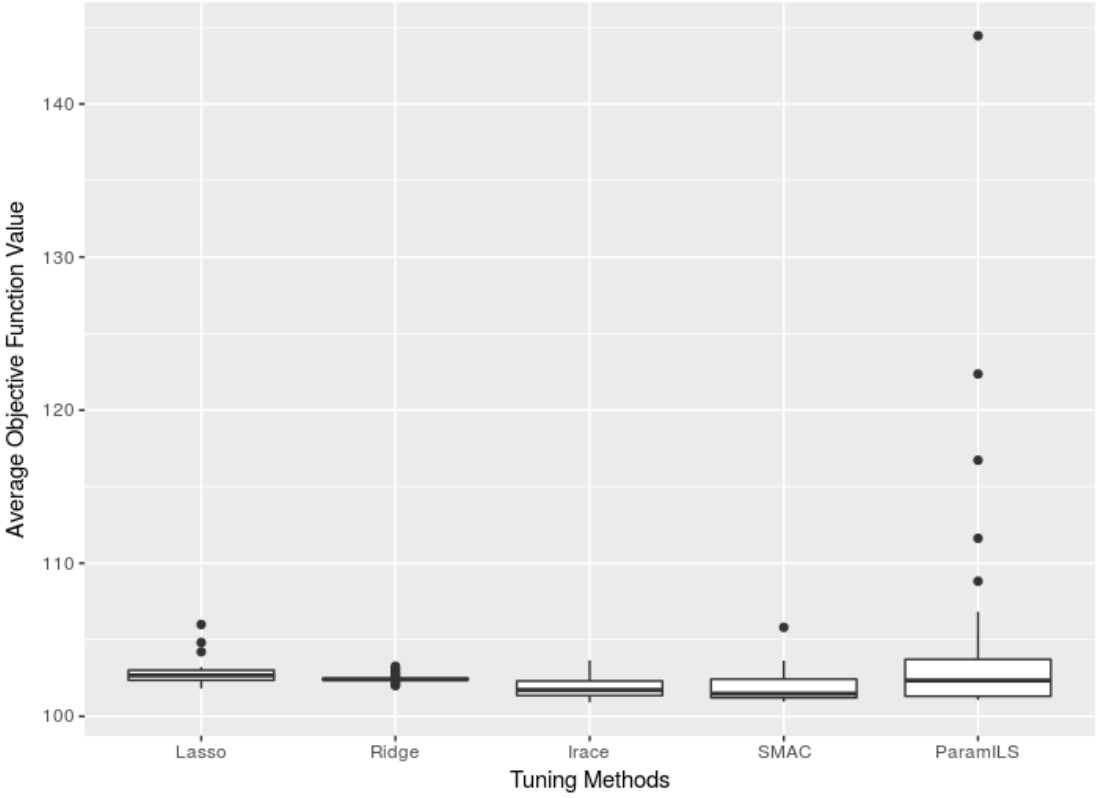

FIGURE 4.4: Overall mean performance - Heterogeneous scenario. Linear and Quantile versions of MetaTuner were omitted due to the presence of extreme outliers, which suggest that these versions can sometimes fail strongly, and may therefore not be interesting for general use.

TABLE 4.3: Mean runtimes for the DE tuning experiment.

Method	**Mean runtime (seconds)**	
	Homogeneous	**Heterogeneous**
Irace	719	265
Ridge	797	741
Lasso	762	352
Linear	730	264
Quantile	724	265
ParamILS	25256	15677
SMAC	24941	18431

In terms of runtime, Irace was marginally faster than the Metatuner versions, and among the MetaTuner versions, the Ridge version seems to be the most time consuming (what can be observed from the table 4.3). Despite these facts, a point mentioned in the discussion of Experiment 1 is reinforced: namely that as the computational cost of evaluating the algorithm being tuned increases, the additional burden of building regression models tends represent a smaller portion of the total computational effort, and consequently the tuning times start becoming less dissimilar.

ParamILS and SMAC had considerably worse mean runtimes for this experiment, but as

with all time considerations this is much more an effect of implementation details than of specific computational efficiency: unlike MetaTuner and Irace (which are both native to R language), the implementations of these two methods had to, at every algorithm evaluation, call an external R script to load and run the optimizer (the DE algorithm, as a R package), which added considerable computational overhead.

Besides analyzing the average performance and runtime of parameter tuning methods, another interesting aspect to investigate is the general distribution of parameter values obtained by the methods. Figures 4.5 and 4.6 illustrate the best parameter values found by the tuning methods (the parameter values are in their original scales). These results suggest the use of reasonably low values of F for both scenarios, but also indicate a large spread of values returned by CR and especially of K.

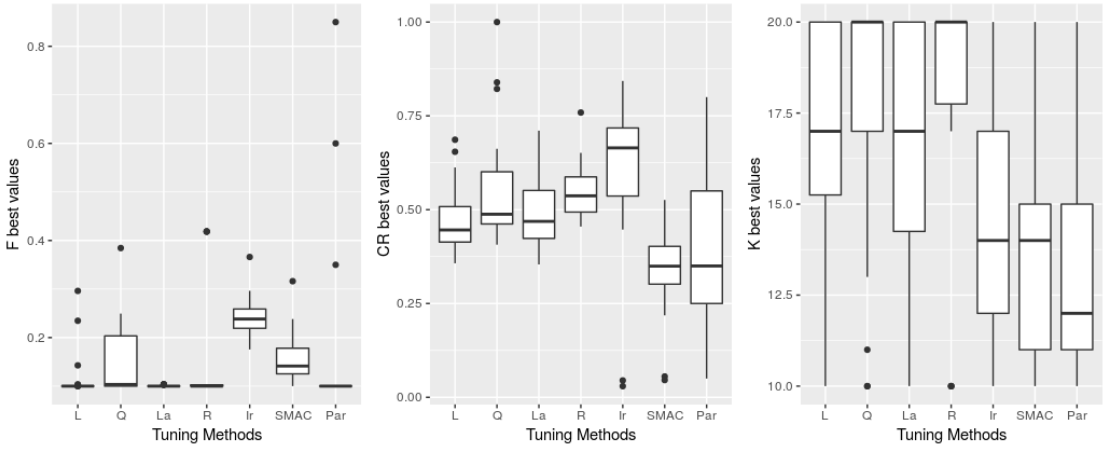

FIGURE 4.5: Distribution of the best parameter values - Homogeneous scenario. The versions of MetaTuner are labelled as: L - Linear; Q - Quantile; La - Lasso; R - Ridge. Irace is labelled as "Ir", and ParamILS as "Par"

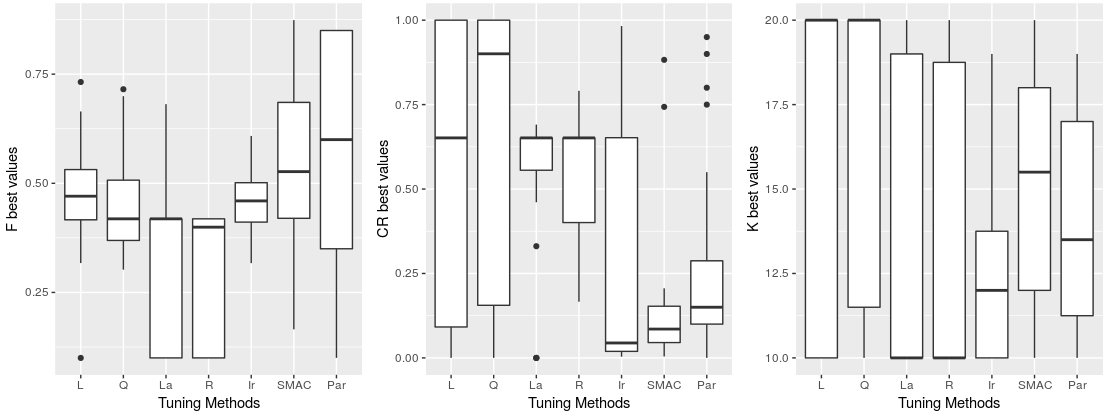

FIGURE 4.6: Distribution of the best parameter values. The parameter tuning methods are labelled as in the prior figure.

As in the prior experiment, the information provided by Metatuner about the algorithm behavior were used to investigate which parameters are the most important contributors

to a given algorithm's performance on a problem class, which is illustrated in the table 4.4. The columns of this table indicate which model parameters were interpreted by MetaTuner as "the most relevant" most often (**Freq#1**), second most-often (**Freq#2**) and third most-often (**Freq#3**). The two numbers in brackets represent how many times the parameter was the most relevant, followed by how many times it was among the 3 most relevant ones.

Table 4.4 suggests that (considering the different combinations of exponents), the MetaTuner versions detected mainly the main effects of CR (29 times as the most relevant and 69 times among the 3 most relevant), F (25 times as the most relevant and 37 times among the 3 most relevant), and the interaction of $F \times CR$ (38 times as the most relevant and 94 times among the 3 most relevant) as possibly the main contributors to the performance of DE/rand/1/bin on the homogeneous scenario. For heterogeneous scenario, the results suggest the main effect of CR (54 times as the most relevant and 113 times among the 3 most relevant), and the interaction of $F \times CR$ (51 times as the most relevant and 85 times among the 3 most relevant) as the main contributors to the performance of DE/rand/1/bin.

In fact, the influence of F, CR and of the interaction of $F \times CR$ showed by MetaTuner echoes in the literature, in several works, when using several versions of DE for solving challenging problems. In [85] are presented visual "maps" showing the quality of DE solutions as a function of choice of F and CR values, and this choice can vary according to the DE version and the problem class. A parameter combination framework for DE is proposed in [86] which employs a strategy of combining different regions of the parameters space of F and CR in order to improve results of DE. In [87] is presented the best 63 combinations of values of F and CR determined by a self-adaptive DE for solving three well known benchmark problem sets. Another self-adaptive variation of DE is presented in [88], and in this work was showed experimentally the influence of different regions of the parameters space of F and CR for the quality of DE solutions, for several challenging problems.

Another important issue is that, at least considering the values used of K $\{10,\ldots,20\}$ the population size multiplier K does not seem to appear prominently as the most relevant factor, which suggests that as long as the computational budget is maintained the population size can be regarded as secondary in comparison to a good selection of F and CR. According to [89], several strategies of choosing the population size, whether or not related to the size of problem have been proposed, but it is not clear the impact of each of them in general.

Summaring, the results of this experiment shows MetaTuner as an advantageous parameter tuning tool in relation to the other methods. It is a competitive approach in

TABLE 4.4: Most relevant terms of DE/rand/1/bin.

Scenario	Version	Freq #1	Freq #2	Freq #3
Homogeneous	Linear	$F \times CR$ [14;27]	CR^2 [13;22]	$F \times CR^2$ [1;19]
	Quantile	$F \times CR$ [13;22]	CR^2 [8;14]	F^2 [7;12]
	Lasso	F [18;25]	CR^3 [6;23]	CR [2;10]
	Ridge	$F \times CR \times K$ [14;20]	$F \times CR$ [8;18]	$F \times CR^2$ [2;8]
Heterogeneous	Linear	CR^2 [18;25]	$F \times CR$ [6,15]	$F \times CR^2$ [2;14]
	Quantile	CR^2 [21;25]	F^3 [7;16]	$F \times CR^2$ [6;13]
	Lasso	$F \times CR^2$ [12;16]	CR^3 [11;24]	CR [3;20]
	Ridge	$F \times CR^2$ [25;27]	$F \times CR \times K$ [3;21]	CR [1;19]

relation to other parameter tuning methods both on Homogeneous or Heterogeneous scenarios, considering the best configurations returned. In addition to this, MetaTuner explicitly models the algorithm performance as a function of the tunable parameters, and it enables the identification of the most relevant contributors to the success of a given algorithm, providing researchers with interesting analyses that can be used to guide algorithm development and adaptation. This latter ability is a clear advantage of MetaTuner in relation to the other parameter tuning methods used here.

4.4 Tuning SAPS for the SAT problem

This experiment is based on guidelines by Montero *et al.* [59], and its main objective is a comparison of MetaTuner with well known methods in cases where few tuning instances are available.

The same four instantiations of MetaTuner used in the prior experiments were used and compared with Irace, SMAC and ParamILS, as well as against a random sampling approach used to provide a performance baseline. The methods were used for tuning the parameters of the Scaling and Probabilistic Smoothing (SAPS) algorithm [90, 91] for solving the SAT problem. Four parameters were tuned: $\alpha \in [1.01, 1.4]$, $wp \in [0, 0.06]$, $\rho \in [0, 1]$, and $ps \in [0, 0.2]$. The ranges were discretised to seven equally-spaced values for ParamILS.[8] Only 10 instances were available[9], and all of them were used for both training and validation. Although this may result in some overfitting of the resulting configurations to the instances, this is an unfortunate consequence of the very limited number of available instances for tuning.

The performance measurement used for each SAPS configuration on each instance was the time-to-convergence, with a timeout of 15 seconds. Cases in which the algorithm failed to converge within that time received a performance value calculated as

[8] Following recommendation from http://www.cs.ubc.ca/labs/beta/Projects/ParamILS/index.html, from where the instances were obtained.

[9] All instances are satisfiable examples of the SAT problem.

$(15.00001 + min(max(0, sol/100000), 0.001))$, where sol is the lowest number of false clauses found. The computational budget used for the tuning process was 1250 evaluations, for all tuning methods (this budget was chosen after preliminary tests using ParamILS and SMAC, when trying keep the total runtime reasonable, considering the cost of tuning processes), and the other initial parameters of all parameter tuning methods were set as in the prior experiment. The random sampling method consisted of randomly generating 125 configurations and running each one once on each instance, returning the configuration with the best median performance. The seeds used for running the SAPS algorithm on instances were the same, for all tuning methods. This experiment was performed in the same machine that was used for the first experiment.

Ten independent replicates were run for each tuning method. The best configurations returned by each method were then run 30 times on each instance (using seeds for the random number generator different from those used during the tuning process), and the overall performance of each configuration was calculated as the average of all means of the observed performances on the instances.

Figure 4.7 presents the distribution of the mean overall performance of the tuning methods, considering the 10 best configurations returned by each one. It is clear from the figure that all tuning methods actually provide better configurations than would be obtained by simply performing a random search over the space of parameters. However, two MetaTuner approaches, namely: the Linear and Quantile, like in the Heterogeneous scenario of the prior experiment, presented outlying replicates in which their performance were worse than that of random search. Even if rare, this is enough to generate scepticism about these two alternatives.

A Kruskal-Wallis test was able to detect the differences as statistically significant at the 95% confidence level ($p = 7.7 \times 10^{-6}$), and the subsequent pairwise Wilcoxon-Mann-Whitney tests indicated ParamILS and Irace as the best, followed by: Lasso, Ridge and SMAC; Quantile and Linear; and Random approach as the worst. Table 4.5 displays the mean runtime of all approaches. As in the previous experiment, as the computational cost of evaluating the algorithm becomes larger, the differences in added computational burden of the methods become less important. Among the MetaTuner versions, the Ridge version seems to be the most time consuming.

Figure 4.8 illustrates the distribution of the parameter values obtained at the end of the 10 replicates for all methods. All of them were able to generate solutions with a large spread of values, suggesting that high-quality configurations for this particular problem possibly emerge from the interaction of parameter values, instead of being dominated by a few main effects of the parameters. Table 4.6 shows the most relevant parameters returned by the versions of MetaTuner for this scenario. Parameters ps and ρ seem to

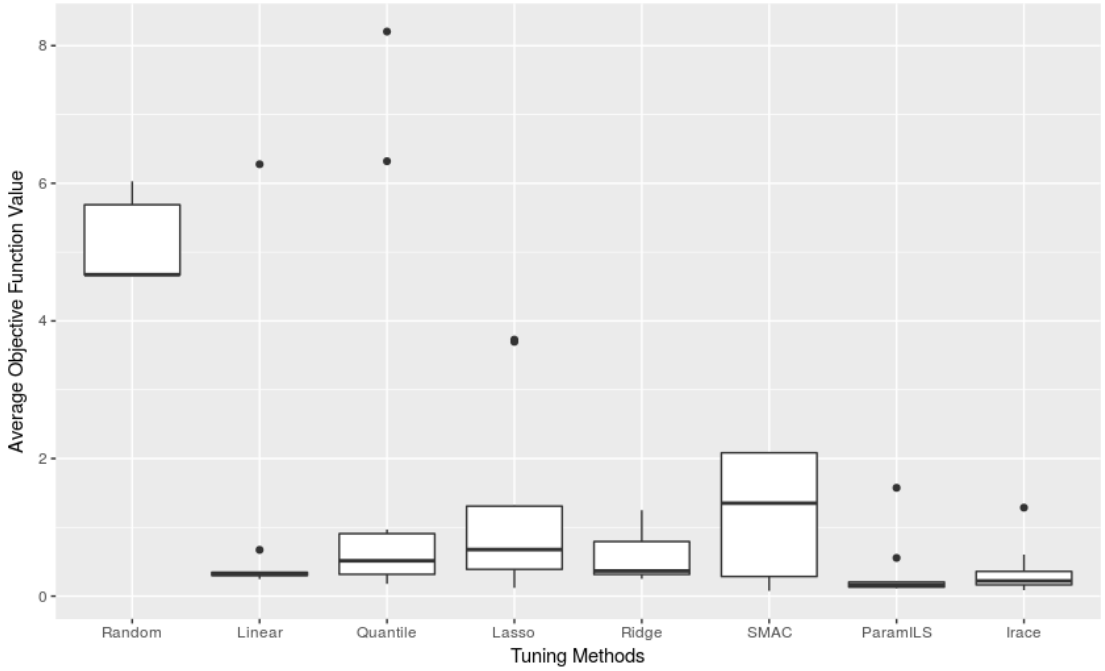

FIGURE 4.7: Overall mean performance of each tuning method for the SAPS algorithm.

TABLE 4.5: Mean runtimes for the SAPS tuning experiment.

Method	**Mean runtime (seconds)**
Ridge	6449
Lasso	4908
Linear	3894
Quantile	4252
Irace	3534
ParamILS	4912
SMAC	3051
Random	9503

appear as the most relevant more often, but there is generally a more heterogeneous distribution of parameters identified as the most relevant ones, which can indicate that all parameters share a similar importance in terms of determining the performance of the algorithm.

From the literature, in [90] is presented an approach called "RSAPS" (Reactive Saps), which is a self-adaptive SAPS. The results of this work suggest that the search intensification of SAPS can be dynamically improved by adapting the values of ρ and ps while fixing values for wp and α, reaching better results than the original SAPS. Although a more intensive research is necessary about this issue, it suggests that ρ and ps can have important influence on the quality of solutions provided by SAPS, as indicated by MetaTuner. As in the prior experiment, MetaTuner is the only one parameter tuning method used here that is able to provide this sort of insight about the relation between algorithm parameters relevance and solution quality for a problem class.

TABLE 4.6: Most relevant parameters - SAPS

Version	**Freq #1**	**Freq #2**	**Freq #3**
Linear	ps^2[6;10]	ρ[2;3]	$wp \times \rho$[1;3]
Quantile	ps^2[6;7]	α^2[1;3]	wp^2[1;2]
Lasso	ρ^3[4;7]	ρ[4;5]	$\alpha \times \rho^2$[2;6]
Ridge	$wp^2 \times \rho$[5;9]	$wp \times \rho \times ps$[3;5]	ρ^3[1;4]

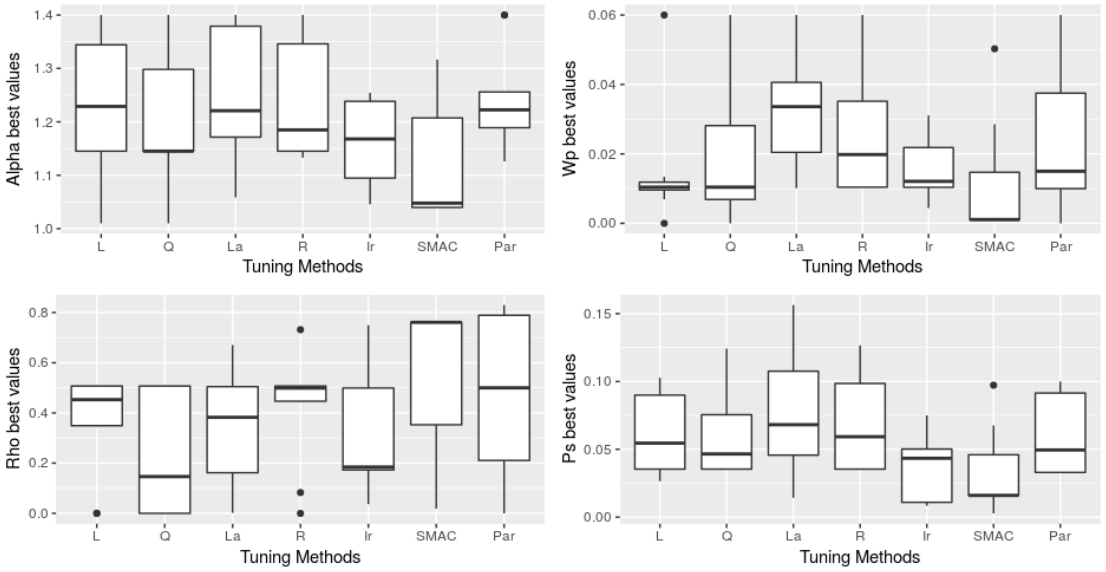

FIGURE 4.8: Distribution of parameter values obtained for the SAPS problem. The versions of MetaTuner are labelled as: L - Linear; Q - Quantile; La - Lasso; and R - Ridge. Irace is labelled as "Ir", and ParamILS as "Par"

4.5 Experiments Overview

With the objective of evaluating the utility of the proposed framework, the experiments performed can be used as a basis for discussing some issues. The variety of the experiments allows to compare the proposed framework against others with respect some interesting factors: when using it for tuning algorithms, considering: extreme algorithm behaviors, with different or similar instance problems, or in a presence of few available instances.

Analyzing the experiment 1 is clear that the versions of MetaTuner is able "to capture" the algorithm behavior, in a way that the closer the model used (for representing the algorithm behavior) to the actual algorithm behavior, better is the result, in terms of parameters relevance. It can be argued that usually the algorithm behavior is unknown, and, therefore is difficult to the user to choose an order of the model. In this sense, is reasonable consider that the response surfaces which correlate the algorithm performance and the parameter values are not highly multimodal (because otherwise it would indicate a somewhat random influence of the parameters, which is not expected).

Thus, having a moderate upper bound for the order of the model (4 or 5, for instance) MetaTuner can be used for searching for the model which fits better the data, starting with one model of order 2. The versions Lasso and Ridge in this case have the advantage that they can highlight the most relevant parameters. Moreover, according to the results of experiment 1, they have a slightly better result for identifying the parameters relevance.

The other 2 experiments showed that the versions Ridge and Lasso have a stable behavior in relation to the quality of the best configurations (considering similar or different problem classes or few instances available), unlike the versions Quantile and Linear. In general, the information about relevance for this 2 experiments are in agreement with works from the literature. These results lead to state that the versions Lasso and Ridge are recommended to be used rather than Quantile and Lasso.

Lastly, and advantage already cited of MetaTuner is its algorithm behavior information. In this context, it allows to have insights of how the parameters influence the algorithm performance, and if eventually there are parameters that not have influence. These information can be used when testing a new optimization algorithm (to verify if the parameters have the expected relevance) or for improving an existent one. Moreover, considering the common problem-dependence of the algorithms, the model algorithm behaviors can be used to learn about how the parameters contribute in different ways for different problems.

Chapter 5

Conclusions and Future Works

5.1 Conclusions

This work is focused on proposing a framework for formulating a new parameter tuning method. This framework is based on concepts from Sequential Model Based Optimization (SMBO) methods. The proposed framework is centred on the sequential optimization of perturbed regression models of expected algorithm performance conditional on parameter values, and on the sequential evaluation of new problem instances on the most promising candidate configurations.

It is proposed to be, at the same time, a tool for: **a)** reaching good parameter values and; **b)** providing to the user regression models which can identify the most relevant parameters, in terms of the main or interaction effects. It was tested in three different experiments, and the main conclusions that can be drawn are: (i) in general, Metatuner is able to yield competitive parameter values when compared with those obtained by other well known parameter methods, in a variety of problem scenarios; and (ii) the proposed method can suggest the most relevant parameters when dealing with a tuning scenario for which the relation between the algorithm performance and the parameter values is dominated by few main and interaction effects of the parameters.

As was already presented by the results of [62] and [59], the experiments performed in this work also showed that it is not possible to state one parameter tuning method as the best one, considering the several factors which can influence on the task of finding the best parameter values: a set of similar or different problem classes to be solved, number of parameters to be tuned, number of available training instances, response surface underlying the correlation betwen algorithm solution quality and parameter values, etc. Beside that, the third experiment showed that using a parameter tuning method instead

of a random search by finding the best parameter values can be an advantage in order to improve the solution quality returned by an algorithm.

Considering the output returned by the parameter tuning methods used here, MetaTuner can offer to the user a more useful output, when compared against the other parameter tuning methods. While all methods used here are useful to that kind of user who wants to compare his/her algorithm against others; only MetaTuner can provide insights that allow the user to explore and analyze the influence of the algorithm parameters in relation to the problem classes, which can help him/her to improve the algorithm design. If it is considered REVAC, which is a parameter tuning method which can return this kind of output, MetaTuner has one advantage: it can reveals the relevance of interaction effects, while REVAC is able to return only the main effects of the parameters. Of course, it is recommended the intensive use of MetaTuner for the task of analyzing the parameters relevance, that is, it must be run many times in order to observe the relevance patterns returned by the whole set of runs. Therefore, it is a more time consuming task than using it just for finding the best parameter values.

Concluding, although automatic parameter tuning tools are not a novelty in the optimization area, MetaTuner brings some contributions to the development of researches in this area: in terms of modelling the algorithm behavior (the main task of the SMBO methods), MetaTuner is the unique tool that provides a set of regression modelling (OLS, Quantile, Lasso and Ridge) to be used. The user, therefore, in face of the optimization scenario (similar problems, different problems, non-normal data, etc) can choose what approach is more suitable to use, or even compare the results of all them, in order to have a more comprehensive set of results. Specifically about the issue of understanding more about the metaheuristics, MetaTuner can help the users to find out which are the relations of the performance of their algorithms and the parameters used, helping them to improve the design of their approaches. The results obtained so far have been recently published in the Applied Soft Computing journal ([92]). However, it is imperative to analyze which are the weakness and restrictions of MetaTuner so that it can be continuously improved, as will be described in the next section.

5.2 Future Works

The parameter tuning framework presented in this work, named MetaTuner is a suitable parameter tuning framework to deal with a small and medium (2 to 8) number of numeric parameters. Although it can be adopted for a large number of parameter tuning tasks, its limitations in relation to the number (because the task of obtaining regression model for several parameters may be proibitive, in terms of runtime) and the type of parameters

can restrict its use. Beside that, in this work only one approach (Nelder-Mead) was used as the algorithm in the optimization regression models phase, and the second and third experiment showed convergence problems with the versions Quantile and Linear. In face of these facts, future works naturally can be proposed.

Notwithstanding MetaTuner has achieved satisfactory results, it is necessary to improve its skills for using it in a broader number of optimization scenarios. The regression modelling options available in MetaTuner do not deal with non numerical parameters. It is a restriction that must be overcome, once it is common parameter tuning tasks with a mixed type of parameters, like parameter tuning of solvers. In this sense, the adaptation of its regression models to deal with categorical values, or the use of other modelling approaches can be investigated, as regression trees or random forests.

Another issue that can be considered is the use of alternative algorithms for optimizing the regression models, with an analysis of using different combinations of optimizers and regression models in relation to two aspects: the quality of the best parameter values and the pattern of relevance parameters. Finally, as presented in the first experiment, the performance of MetaTuner can be damaged as the increasement of the number of parameters. This fact points out to the necessity of using options of modelling more insensitive to the number of parameters. Lastly, an investigation of the convergence problems with the versions Quantile and Linear is mandatory.

Appendix A

Regression Modeling

A.1 Regression Models

To explain the general concepts of regression modeling, the simplest version of these tools, namely the *linear regression model*, will be used. To illustrate this, suppose one wish to derive an empirical model relating two input variables, pressure and temperature, to a response variable, e.g., the speed of a chemical reaction [1]. A possible model structure to describe this relationship is:

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \epsilon \tag{A.1}$$

where y represents the reaction speed, x_1 represents the pressure and x_2 represents the temperature. The input variables x_1 and x_2 are called *independent variables* or *regressor variables*. The variable y is the *response variable*. The coefficients β_0, β_1 and β_2 are the *regression coefficients*. This model is called *Linear regression* as the equation A.1 is linear in relation to β_0, β_1 and β_2. Finally, ϵ is the variable associated with the modelling error, and encapsulates the effects of experimental error and other unmodelled effects.

This model defines a plane or a *response surface* in the two-dimensional space, in which β_0 is its intercept, β_1 measure how the value of y will change when x_1 is changed by 1 unit, and the same idea is stated for β_2 and x_2.

More generally, a regression model can be built using k independent variables, obtaining the equation [63]:

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \ldots + \beta_k x_k + \epsilon \tag{A.2}$$

[1] this example is based on one presented in the Design and Analysis of Experiments Book [63]

The equation A.2 defines a *multiple linear regression* with k regressor variables and $k+1$ regression coefficients. Depending on the nature of the relationship between the regressor variables and the response variable, the model can assume the form of a response surface of second or higher order. The following example shows a multiple linear regression model that generates a second-order response surface, according to [63]:

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_{11} x_1^2 + \beta_{22} x_2^2 + \beta_{12} x_1 x_2 + \epsilon \tag{A.3}$$

Using regression models for approximating the behavior of an experiment is suitable if the coefficients β_0,...β_k are estimated by minimizing the error between the sampled values y of the response variable; and the predicted values $\hat{y}$ by applying the model to the known values of the regressor variables. This estimation can be done tipically using the minimization of the squared erros. Considering the model described in the equation A.2 and that we have $n > k$ observations of y, then each value y_i is modeled as:

$$y_i = \beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + \ldots + \beta_k x_{ik} + \epsilon_i \tag{A.4}$$

A.1.1 Ordinary Lest Square (OLS) Regression

Consider that the experimental error ϵ in equation A.4 is an independent random variable with expected value $E(\epsilon) = 0$ and variance $V(\epsilon) = \sigma^2$. For each observation the error ϵ_i is given as:

$$\epsilon_i = y_i - \beta_0 - \sum_{j=1}^{k} \beta_j x_{ij} = y_i - \hat{y}_i \tag{A.5}$$

The total squared error for n observations of y can be calculated as:

$$\epsilon^2 = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 \tag{A.6}$$

Thus, a basic strategy for defining the β coefficients is to find the values of these model parameters that minimize the squared error, i.e., by finding the *least squares estimates*, and this regression modeling is called *Ordinary Least Square (OLS) Regression*. The vector of least squares estimates $\hat{\beta} \in \mathbb{R}^{k+1}$ can be calculated as [63]:

$$\hat{\beta} = (X'X)^{-1} X'y \tag{A.7}$$

where $X \in \mathbb{R}^{n\times(k+1)}$ is a matrix of observations of the independent variables (with a first column set to unity added, so that the intercept term β_0 can be fit):

$$X = \begin{bmatrix} 1 & x_{11} & \dots & x_{1k} \\ 1 & x_{21} & \dots & x_{2k} \\ \vdots & \vdots & \vdots & \vdots \\ 1 & x_{n1} & \dots & x_{nk} \end{bmatrix}$$

and $\mathbf{y} \in \mathbb{R}^{(n)}$ is the vector of observations of the dependent variable.

The estimators $\hat{\beta}_0, \hat{\beta}_1, \dots, \hat{\beta}_k$ describe an equation which can predict (within a certain level of confidence) new values of the response variable y when new values of $x_1, x_2, \dots x_k$ are given. More details are given in [63].

A.1.2 Quantile Regression

An arbitrary value y_i of a random variable Y scores at the $\tau - th$ quantile if it is greater than the proportion τ of values from Y and smaller than the proportion of $(1-\tau)$. In this way, the quantile of $\tau = 0.5$ represents the median of Y, dividing the population into two equal segments. Similarly, the quartiles divide the population into four equal proportions of the population. Percentiles and fractiles refer to general case ([72]). More formally, any real valued random variable Y may be characterized by its distribution function ([93]):

$$F(y) = Prob(Y \leq y) \tag{A.8}$$

and for any $0 < \tau < 1$,

$$Q(\tau) = \inf\{y : F(y) \geq \tau\} \tag{A.9}$$

is called the $\tau - th$ quantile of Y. Whereas the *OLS* regression aims to find a model (equation) that describes the mean behavior of a response variable Y, as a function of the independent variables $\{x_1, ...x_n\}$; the quantile regression of τ determines a model which describes the behavior of Y as a function of the independent variables $\{x_1, ...x_n\}$, in a way that the probability of the observed values y_i to be $\leq \hat{y}_i$ (its predicted value related) is $\geq \tau$. For instance, a quantile regression with $\tau = 0.5$ obtains a model which represents the median behavior of Y as a function of the independent variables.

Quantile regression can be obtained by solving an optimization problem ([93]). For any $0 < \tau < 1$, define the piecewise linear "check function":

$$\rho_\tau(u) = u(\tau - I(u < 0)) \tag{A.10}$$

where I is the indicator function and u is the prediction error. Minimizing the expectation of $\rho_\tau(Y - \xi)$ with respect to ξ yields solution $\hat{\xi}(\tau)$, the smallest of $Q(\tau)$. Considering a random sample of Y, $\{y_1, \ldots, y_n\}$, the *sample quantile* $\tau - th$ may be found by solving:

$$\min_{\xi \in R} \sum_{i=1}^{n} \rho_\tau(y_i - \xi) \tag{A.11}$$

The equation above defines the *unconditional quantiles* as an optimization problem. In order to obtain the *conditional quantiles*, that is, a function of independent variables and regression coefficients, the scalar ξ is replaced by a parametric function $\xi(x_i, \beta)$:

$$\min_{\beta \in \Re^p} \sum_{i=1}^{n} \rho_\tau(y_i - \xi(x_i - \beta)) \tag{A.12}$$

Being $\xi(x, \beta)$ formulated as a linear function of parameters, the problem can be solved by linear programming methods. Quantile regression is especially useful because its robustness to distributional assumptions of the response observations ([93]). It works well, regardless of the distribution of response data. In these scenarios, in which *OLS* regression can degenerate its estimators of regression coefficients because of having outliers, quantile regression obtains more reliable ones. However, it can be quite sensitive to outliers in the independent variables ([93]). Despite of it, it has been widely used in economy, ecology, medicine, etc ([93], [94]).

A.1.3 Ridge and Lasso Regression

According to Tibishirani ([95]), there are two reasons why the data analyst is often not satisfied with the *OLS* regression: *prediction accuracy* and interpretation. Poor *prediction accuracy* can be caused by a low bias and large variance of *OLS* regression. Thinking about interpretation, a large number of predictors (regression coefficients) can become it difficult. Shrinking or setting to 0 some coefficients can minimize this problem.

Shrinking the coefficients can be achieved by adding a penalty term in the problem of minimizing the least squared errors. Considering the predictor of the response variable y

as a linear function of the form $\beta_0 + x^T\beta$ (with $\beta \in \Re^p$), the problem becomes (adapted from [96]):

$$\min_{\beta\in\Re} \sum_{i=1}^{n}(y_i - \beta_0 - x_i^T\beta)^2 + \lambda\sum_{j=1}^{p}\beta_j^2 \tag{A.13}$$

where:

β_0: is the intercept of the equation
λ: is the regularization parameter, with $\lambda \in [0, \infty]$

As λ increases, increases the shrinkage of coefficients and the bias of the model, but reducing its variance. The use of this regression approach may be important in the presence of a large number of regression coefficients, with substantial differences of relevance between them. Shrinking them will reduce all regressor values, leading those with least relevant to values near to zero, boosting the relative difference between them and helping the model interpretation. Of course, this task is dependent of λ parameter, and the proper choose of this value is itself an optimization problem.

Another kind of regression shrinkage is called *lasso* (least absolute shrinkage and selection operator, [95]). It shrinks some regression coefficients and sets others exactly to 0, trying to select those which are more relevant. Lasso is formalized modifying the equation A.13 to the following:

$$\min_{\beta\in\Re} \sum_{i=1}^{n}(y_i - \beta_0 - x_i^T\beta)^2 + \lambda\sum_{j=1}^{p}|\beta_j| \tag{A.14}$$

The effect of the linear penalty imposed by the equation A.14 is that possibly the solution of the problem will contain some coefficients equal to 0, promoting the coefficient selection. Some improvements to ridge and lasso has been presented: Owen ([96]) presented a robust hybrid of lasso and ridge regression, in which he replaced the minimization of least squares by minimizing the Huber's criterion, which minimize the squared errors when dealing with relatively small errors, and the absolute errors in the case of large errors. Arslan ([97]), presented a Weighted LAD-LASSO method, which consists in minimizing the weighted absolute errors with a linear penalty. According to the author, this method is robust in terms of outliers, leverage points (outliers in the independent variables) and is capable to promote the selection of coefficients.

Yi and Huang ([98]) proposed a method of minimizing a robust function loss (Hubber or Quantile functions) subject to a hybrid penalty term (composed by a L1-norm and

a L2-norm components) by implementing a semismooth Newton coordinate descent. Choosing suitable weights to the penalty components, it is possible to use this method as a Lasso, Ridge or a mixed Lasso-Ridge regression. This latter work is used as a basis to implement the R Package *hqreg* used in this framework. More detailed, the optimization problems represented by the equations A.13 and A.14 are combined as follows:

$$\min_{\beta \in \Re} \sum_{i=1}^{n} l(u) + \lambda P(\beta) \tag{A.15}$$

where $l(u)$ is a function loss and $P(\beta)$ is a penalty function, with $\lambda \geq 0$.

By default, the function loss used is the Huber loss, described as (adapted from [98]):

$$l(u) = h_{\gamma}(u) = \begin{cases} \frac{u^2}{2\gamma}, \text{if } |u| \leq \gamma \\ |u| - \frac{\gamma}{2}, \text{if } |u| > \gamma \end{cases} \tag{A.16}$$

where γ is the tuning parameter for Huber loss, defined by default as being equal to $IQR/10$, with IQR equal to the data interquartile range. For selecting an optimal value of λ, it is performed a cross validation (with 10 cross-validation folds by default) for the penalty Huber loss regression over a sequence of lambda values.

The penalty function is defined as follows:

$$P(\beta) = P_{\alpha}(\beta) = \alpha||\beta||_1 + (1-\alpha)\frac{1}{2}||\beta||_2^2, \text{ with } 0 \leq \alpha \leq 1 \tag{A.17}$$

According to the equation A.17, if α is 0 the ridge regression is performed, and if α is equal to 1 the lasso regression. All default values used by the *hqreg* R package for its parameters are adopted by this framework.

Bibliography

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. 2th edition edition, 2001.

[2] M. Birattari. *Tuning Metaheuristics - A Machine Learning Perspective*. Springer-Verlag Berlin Heidelberg, 1ª edition, 2005. doi: 10.100/978-3-642-00483-4.

[3] A.R. Trindade and L. S. Ochi. Um algoritmo evolutivo híbrido para formação de células de manufatura em sistemas de produção. *Pesquisa Operacional*, 26(2): 255–294, 2006.

[4] A. K. Masum, Md. F. Faruque, M. Shahjalal, and Md. I. H. Sarker. Solving the vehicle routing problem using genetic algorithm. *IJACSA - International Journal of Advanced Computer Science and Applications*, 2(7), 2011.

[5] J. Y. Potvin. Genetic algorithms for the traveling salesman problem. *Annals of Operations Research*, 63:339–370, 1996.

[6] A. N. Unal. A genetic algorithm for the multiple knapsack problem in dynamic environment. *World Congress on Engineering and Computer Science - WCECS Vol II*, 2013.

[7] B. M. Baker and M. A. Aechew. A genetic algorithm for the vehicle routing problem. *Computers & Operations Research*, 30:787–800, 2003.

[8] J. N. Hooker. Needed: An empirical science of algorithms. *Operations Research*, 42 (2):201–212, March-April 1994.

[9] J. N. Hooker. Testing heuristics: We have it all wrong. *Journal of Heuristics*, (1): 33–42, 1995.

[10] G. Karafotias, M. Hoogendoorn, and A. E. Eiben. Parameter control in evolutionary algorithms: Trends and challenges. *IEEE Transactions on Evolutionary Computation*, 19(2):167–187, April 2015.

[11] R. Battiti and G. Tecchiolli. The reactive tabu search. *ORSA Journal on Computing*, 6(2):126–140, 1994.

[12] R. Battiti. Reactive search: Toward self-tuning heuristics. In C. R. Reeves V. J. Rayward-Smith, I.H. Osman and G. D. Smith, editors, *Modern Heuristic Search Methods*, chapter 4, pages 61–83. John Wiley and Sons Ltd, 1996.

[13] R. Battiti and M. Brunato. Reactive search: Machine learning for memory-based heuristics. In T. F. Gonzales, editor, *Handbook of Approximation Algorithms and Metaheuristics*, pages 21.1–21.17. Chapman & Hall/CRC, Boca Raton, 2007.

[14] K. Liang, X. Yao, and C. Newton. Adsearch self-adaptive parameters in evolutionary algorithms. *Applied Intelligence*, 15(3):171–180, 2001.

[15] A. E. Eiben, R. Hinterding, and Z. Michalewicz. Parameter control in evolutionary algorithms. *IEEE Transactions on Evolutionary Computation*, 3(2), July 1999.

[16] M. Prais and C. C. Ribeiro. Reactive grasp: An application to a matrix decomposition problem in tdma traffic assignment. *INFORMS Journal on Computing*, 12: 164–176, 2000.

[17] Md. Asafuddoula, T. Ray, and R. Sarker. A self-adaptive differential evolution algorithm with constraint sequencing. In *Australian Joint Conference on Artificial Intelligence*, pages 182–193, 2012.

[18] J. Sun, W. Xu, and B. Feng. Adaptive parameter control for quantum-behaved particle swarm optimization on individual level. In *IEEE International Conference on Systems, Man and Cybernetics*, pages 420–428, 2005.

[19] J. Brest, V. Zumer, and M. S. Maucec. Self-adaptive differential evolution algorithm in constrained real-parameter optimization. In *IEEE International Conference on Evolutionary Computation*, pages 182–193, 2006.

[20] W.-J. Yu, M. Shen, W.-N. Chen, Z.-H. Zhan, Y-J. Gong, Y. Lin, O. Liu, and J. Zhang. Differential evolution with two-level parameter adaptation. In *IEEE Transactions on Cybernetics*, volume 44, pages 1080–1099, 2014.

[21] P. Vajda, A. E. Eiben, and W. Hordijk. Parameter control methods for selection operators in genetic algorithms. In *International Conference on Parallel Problem Solving from Nature*, pages 620–630, 2008.

[22] S.-H. Liu, M. Mernik, and B. R. Bryant. Entropy-driven parameter control for evolutionary algorithms. *Informatica*, 31:41–50, 2007.

[23] S.-H. Liu, M. Mernik, D. Hrnčič, and M. Črepinšek. A parameter control method of evolutionary algorithms using exploration and exploitation measures with a practical application for fitting sovova's mass transfer model. *Applied Soft Computing*, pages 3792–3805, 2013.

[24] Q. Lin, Z. Liu, Q. Yan, Z. Du, A. C. Coello, Z. Liang, W. Wang, and J. Chen. Adaptive composite operator selection and parameter control for multiobjeticve evolutionary algorithm. *Information Sciences*, 339:332–352, 2016.

[25] A. Aleti and I. Moser. A systematic literature review of adaptive parameter control methods for evolutionary algorithms. *ACM Computing Surveys (CSUR)*, 49:1–35, 2016.

[26] V. Nannen and A. E. Eiben. Relevance estimation and value calibration of evolutionary aalgorithm parameters. *International Joint Conference on Artificial Intelligence*, January 2007.

[27] B. Adenso-Díaz and M. Laguna. Fine-tuning of algorithms using fractional experimental design and local search. *Operations Research*, 1(54):99–114, 2006.

[28] M. Birattari, T. Stutzle, Yuan Z., and P. Balaprakash. F-race and iterated f-race: An overview. Technical report, IRIDIA - Université Libre de Bruxelles - Institut de Recherches Interdisciplinaires et de Développements en Intelligence Artificielle, June 2009.

[29] S. P. Coy, B. L. Golden, G. C. Runger, and E. A. Wasil. Using experimental design to find effective parameter settings for heuristics. *Journal of Heuristics*, 1(7):77–97, 2001.

[30] A. E. Eiben and S. K. Smit. Parameter tuning for configuring and analyzing evolutionary algorithms. *Swarm and Evolutionary Computation*, 1(1):19–31, 2011.

[31] H. H. Hoos. Automated algorithm configuration and parameter tuning. In *Autonomous Search*, pages 37–71. Springer, 2012.

[32] T. Bartz-Beielstein, M. Chiarandini, L. Paquete, and M. Preuss. *Experimental Methods for the Analysis of Optimization Algorithms.* Springer, Berlin, Germany., 2010.

[33] M. Birattari. *The race package for R: Racing methods for the selection of the best.* Technical Report. TR/IRIDIA/2003-037, IRIDIA, Université Libre de Bruxelles, Belgium, 2003.

[34] M. Birattari, T. Stutzle, L. Paquete, and K. Varrentrapp. A racing algorithm for configuring metaheuristics. In *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2002*, pages 11–18. Morgan Kaufmann Publishers, San Francisco, CA, 2002.

[35] F. Hutter, H. H. Hoos, K. Leyton-Brown, and T. Stutzle. Paramils: an automatic algorithm configuration framework. *Journal of Artificial Intelligence Research*, (36):267–306, October 2009.

[36] M. López-Ibáñez, J. Dubois-Lacoste, T. Stutzle, and M. Birattari. The irace package, iterated race for automatic algorithm configuration. Technical report, Technical Report TR/IRIDIA/2001-4, IRIDIA, Université Libre de Bruxelles, Belgium, 2011.

[37] P. Balaprakash, M. Biratari, and T. Stutzle. Improvement strategies for the f-race algorithm: Sampling design and iterative refinement. *Hybrid Metaheuristics*, Volume 4771 of the series Lecture Notes in Computer Science:108–122, 2007.

[38] M. López-Ibáñez, J. Dubois-Lacoste, L. P. Cáceres, T. Stutzle, M. Birattari, Eric Yuan, and P. Balaprakash. *Package 'irace' - Iterated Racing Procedures*. IRIDIA - Université Libre de Bruxelles, Belgium 2011., november 2014.

[39] S. K. Smit and A. E. Eiben. Multi-problem parameter tuning using bonesa. *Artificial Evolution*, 2011.

[40] T. Bartz-Beielstein, C. W. G. Lasarczyk, and M. Preuss. Sequential parameter optimization. *Evolutionary Computation*, 1:773–780, 2005. doi: 10.1109/CEC.2005.1554761.

[41] F. Hutter, H. H. Hoos, and K. Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *Proceedings of the 5th International Conference on Learning and Intelligent Optimization*, LION'05, pages 507–523, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-25565-6. doi: 10.1007/978-3-642-25566-3_40. URL http://dx.doi.org/10.1007/978-3-642-25566-3_40.

[42] N. Večěk, M. Mernik, B. Filipič, and M. Črepinšek. Parameter tuning with chess rating system (crs-tuning) for meta-heuristic algorithms. *Information Sciences*, 372:446–469, 2016.

[43] D. J. Sheskin. *Handbook of Parametric and Nonparametric Statistical Procedures*. Chapman & Hall/CRC, fifth edition edition, 2011.

[44] W. J. Conover. *Practical Nonparametric Statistics*. third edition edition, 1999.

[45] S. H. Hurlbert. Pseudoreplication and the design of ecological field experiments. *Ecological Monographs*, 54(2):187–211, June 1984.

[46] S. E. Lazic. The problem of pseudoreplication in neuroscientific studies: is it affecting your analysis? *Lazic BMC Neuroscience*, 5(11), 2010.

[47] R. B. Millar and M. J. Anderson. Remedies for pseudoreplication. *Fisheries Research*, (70):397–407, 2004. doi: 10.1016/j.fishres.2004.08.016.

[48] T. Bartz-Beielstein. Sequential parameter optimization. *IEEE Congress on Evolutionary Computation*, pages 773–780, 2009.

[49] E. Haasdijk, A. E. Eiben, and S. K. Smit. Exploratory analysis of an on-line evolutionary algorithm in simulated robots. *Evolutionary Intelligence*, 4(5):213–230, 2012.

[50] F. Hutter, H. Hoos, and K. Leyton-Brown. An evaluation of sequential model-based optimization for expensive blackbox functions. *GECCO'13 Companion*, July 2013.

[51] D. R. Jones, C. D. Perttunen, and B. E. Stuckman. Lipschitzian optimization witwith the lipschitz constant. *Journal of Optimization Theory and Applications*, 79(1):157–181, October 1993.

[52] N. Hansen and A. Ostermeier. Adapting arbitrary normal mutation distributions in evolution strategies: the covariance matrix adaptation. *Proc. of CEC-96*, pages 312–317, 1996.

[53] Will. Ruth and T. Loughin. The effect of heterocedasticity on regression trees. Technical report, Cornell University, June 2016.

[54] Edmund Burke, Graham Kendall, Jim Newall, Emma Hart, Peter Ross, and Sonia Schulenburg. Hyper-heuristics: An emerging direction in modern search technology. In Boston MA Springer, editor, *Handbook of metaheuristics*, pages 457–474. 2003.

[55] F. Caraffini, F. Neri, and M. Epitropakis. Hyperspam: A study on hyper-heuristic coordination strategies in the continuous domain. *Information Sciences*, (477):186–202, 2019.

[56] Shin Siang Choong, Li-Pei Wong, and Chee Peng Lim. Automatic design of hyper-heuristic based on reinforcement learning. *Information Sciences*, (436):89–107, 2018.

[57] Pericles B. C. Miranda, Ricardo B. C. Prudêncio, and Gisele L. Pappa. H3ad: A hybrid hyper-heuristic for algorithm design. *Information Sciences*, 414:340–354, 2017.

[58] V. Nannen and A. E. Eiben. A method for parameter calibration and relevance estimation in evolutionary algorithms. *GECCO'06 - Genetic and Evolutionary Computation Conference*, July, 8-12 2006.

[59] Elizabeth Montero, M-C Riff, and B. Neveu. A beginner's guide to tuning methods. *Applied Soft Computing*, 17:39–51, 2014.

[60] R. Engelke and R. Ewald. Configuring simulation algorithms with paramils. *Proceedings of the 2012 Winter Simulation Conference*, pages 4673–4781, 2012.

[61] A. R. KhudaBukhsh, L. Xu, H. H. Hoos, and K. Leyton-Brown. Satenstein: Autoautomatic building local search sat solvers from components. *Proceedings of the Twenty-First International Joint Conference on Artificial Intelligence (IJCAI-09)*, pages 517–524, 2009.

[62] S. K. Smit. *Parameter Tuning and Scientific Testing*. PhD thesis, Vrije Universiteit, Amsterdam, October 2012.

[63] D. C. Montgomery. *Design and Analysis of Experiments*. fifth edition edition, 2012.

[64] M. J. Crawley. *The R Book*. Wiley, second edition, 2012.

[65] M. D. McKay, R. J. Beckman, and W. J. Conover. A comparison of three methods for selecting value of input variables in the analysis of output from a computer code. *Technometrics*, 21(2):239–245, May 1979.

[66] G. D. Wyss and K. H. Jorgensen. *A User's Guide to LHS: Sandia's Latin Hypercube Sampling Software*. Risk Assessment and Systems Modeling Department - Sandia National Laboratories, February 1998.

[67] S. Joe and F. Y. Kuo. Constructing sobol sequences with better two-dimensional projections. *SIAM Journal on Scientific Computing*, 30(5):2635–2654, 2008.

[68] Kalyanmoy Deb and Samir Agrawal. A niched-penalty approach for constraint handling in genetic algorithms. In *Artificial Neural Nets and Genetic Algorithms*, pages 235–243. Springer-Verlag Science + Business Media, 1999.

[69] Ian H Witten, Eibe Frank, Mark A Hall, and Christopher J Pal. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, 2016.

[70] L. Pérez, M. López-Ibáñez, and T. Stutzle. An analysis of parameters of irace. Technical Report 14, IRIDIA - Institute de Recherches Interdisciplinaires et de Développements en Intelligence Artificielle, November 2013.

[71] M. Birattari. On the estimation of the expected performance of a metaheuristic on a class of instances. how many instances, how many runs? Technical Report TR/IRIDIA/2004-001., IRIDIA, Université Libre de Bruxelles, Belgium., 2004.

[72] Roger Koenker and Kevin F. Hallock. Quantile regression. *Journal of Economic Perspectives*, 15(4):143–156, 2001.

[73] R. Koenker. *Package quantreg*. r-project.org, https://cran.r-project.org/web/packages/quantreg/quantreg.pdf, February 2018.

[74] C. Yi. *hqreg: Regularization Paths for Lasso or Elastic-Net Penalized Huber Loss Regression and Quantile Regression*, 2017. URL https://CRAN.R-project.org/package=hqreg.

[75] A. J. Dobson and A. G. Barnett. *An Introduction to Generalized Linear Models*. 4th edition, 2018.

[76] A. Díaz-Manríquez, G. Toscano, J. H. Barron-Zambrano, and E. Tello-Leal. A review of surrogate assisted multiobjective evolutionary algorithms. *Computational Intelligence and Neuroscience*, (4):1–14, 2016.

[77] Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. An evaluation of sequential model-based optimization for expensive blackbox functions. In *Proc. Genetic and Evolutionary Computation Conference*, pages 1209–1216, July 2013.

[78] D. R. Jones, M. Schonlau, and W. J. Welch. Efficient global optimization of expensive black-box functions. *Journal of Global Optimization*, 13:445–492, 1998.

[79] L. Shi and K. Rasheed. A survey of fitness approximation methods applied in evolutionary algorithms. In Berlin Heidelberg Springer, editor, *Computational Intelligence in Expensive Optimization Problems*, pages 3–28. 2010.

[80] F. Goulart, S. T. Borges, F. C. Takahashi, and F. Campelo. Robust multiobjective optimization using regression models and linear subproblems. In *Proc. Genetic and Evolutionary Computation Conference*, pages 569–576, 2017.

[81] R. Jiao, S. Zeng, C. Li, Y Jiang, and Y. Jin. A complete expected improvement criterion for gaussian process assisted highly constrained expensive optimization. *Information Sciences*, (471):80–96, 2019.

[82] S. M. Mousavi, J. Sadeghi, S. T. A. Niaki, N. Alikar, A. Bahreininejad, and H. S. C. Metselaar. Two parameter-tuned meta-heuristics for a discounted inventory control problem in a fuzzy environment. *Information Sciences*, 276:42–62, 2014.

[83] F. Campelo and Moisés Botelho. Experimental investigation of recombination operators for differential evolution. In *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, GECCO '16, pages 221–228. ACM, 2016.

[84] N. Hansen, A. Auger, S. Finck, and R. Ros. Real-parameter black-box optimization benchmarking bbob-2010: Experimental setup. Technical Report Research report RR-7215, INRIA, 2010.

[85] M. Cervenka and H. Boudná. Visual guide of f and cr parameters influence on differential evolution solution quality. In *24th International Conference on Engineering Mechanics*, pages 141–144, 2018. doi: 10.21495/91-8-141.

[86] Z. Dong J. Zhang. Parameter combination framework for the differential evolution algorithm. *Algorithms*, 12(4):1–22, April 2019. doi: doi:10.3390/a12040071.

[87] T. Ray R. A. Sarker, S. M. Elsayed. Differential evolution with dynamic parameters selection for optimization problems. *IEEE Transactions on Evolutionary Computation*, 18(5):689–707, October 2014.

[88] B. Bošković M. Mernik J. Brest, S. Greiner and V. Zume. Self-adapting control parameters in differential evolution: A comparative study on numerical benchmark problems. *IEEE Transactions on Evolutionary Computation*, 10(6):646–657, 2007.

[89] A. P. Piotrowski. Review of differential evolution population size. *Swarm and Evolutionary Computation*, 32(2017):1–24, 2016.

[90] D. A. D. Tompkins and H. H. Hoos. Scaling and probabilistic smoothing: Dynamic local search for unweighted max-sat. In *Conference of the Canadian Society for Computational Studies of Intelligence*, pages 145–159. Springer, 2003.

[91] D. A. D Tompkins and H. H. Hoos. Ubcsat: An implementation and experimentation environment for sls algorithms for sat and max-sat. In *Theory and Applications of Satisfiability Testing: 7th International Conference, SAT 2004.*, pages 306–320. May 2004.

[92] A. R. Trindade and F. Campelo. Tuning metaheuristics by sequential optimisation of regression models. *Applied Soft Computing*, 85, December 2019. doi: https://doi.org/10.1016/j.asoc.2019.105829.

[93] Roger Koenker. Quantile regression. In Stephen Fienberg and Jay Kadane, editors, *International Encyclopedia of the Social Sciences*. October 2000.

[94] Brian S Cade and Barry R Noon. A gentle introduction to quantile regression for ecologists. *Frontiers in Ecology and the Environment*, 1(8):412–420, 2003.

[95] Robert Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society. Series B (Methodological)*, 58(1):267–288, 1996.

[96] Art B. Owen. A robust hybrid of lasso and ridge regression. Technical report, Stanford University, October 2006.

[97] Olcay Arslan. Weighted lad-lasso method for robust parameter estimation and variable selection in regression. *Computational Statistics and Data Analysis*, 56:

1952–1965, 2012. URL http://www.sciencedirect.com/science/article/pii/S0167947311004208.

[98] Congrui Yi and Jian Huang. Semismooth newton coordinate descent algorithm for elastic-net penalized huber loss regression and quantile regression. Technical report, University of Iowa - Department of Statistics and Actuarial Science, May 2016. URL https://arxiv.org/abs/1509.02957v2. Available at Cornell University Library.