View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by AMS Tesi di Laurea

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

SCUOLA DI INGEGNERIA E ARCHITETTURA
Corso di Laurea Magistrale in Ingegneria Informatica

Combining Active Learning and Mathematical Programming: a hybrid approach for Transprecision Computing

TESI DI LAUREA
in
INTELLIGENT SYSTEMS M

Relatore
Prof.ssa Milano Michela

Presentata da
Bambini Alberto

Correlatore
Dott. Borghesi Andrea

Anno Accademico 2018/19
Sessione III

Summary

Summary ... I

List of tables ... III

List of figures ... V

1 Introduction ... 1

2 Transprecision Computing ... 3

2.1 Concept ... 3

2.2 OPRECOMP ... 5

2.3 FlexFloat ... 6

3 Active Learning ... 9

3.1 Concept ... 9

4 Mathematical Programming ... 13

4.1 CPLEX and DOcplex ... 14

4.2 Empirical Model Learning ... 14

5 Project and objectives ... 17

5.1 Optimization Model ... 18

5.2 Machine Learning Model and Active Learning ... 19

5.3 Benchmarks and Variables Graph ... 21

5.4 Structure ... 25

6 Results ... 39

6.1 Consistency Tests ... 40

6.2 Neighborhood search Tests ... 46

6.3 Active Learning Trend Tests ... 48

7 Conclusions ... 55

References ... 59

Appendices 61
I. Convolution executions 61
II. Correlation executions 66
III. Convolution – failed initial training 71

List of tables

Table 1: Experiment report structure 39

Table 2: Correlation execution data, target 10^{-7} (1) 41

Table 3: Correlation execution data, target 10^{-7} (2) 41

Table 4: Correlation execution data, target 10^{-7} (3) 42

Table 5: Correlation execution data, target 10^{-7} (4) 43

Table 6: Correlation execution data, target 10^{-7} (5) 44

Table 7: Correlation execution data, target 10^{-5}, all neighborhood search versions 47

Table 8: Convolution execution data, target 10^{-15} 49

Table 9: Convolution execution data, target 10^{-20} 51

Table 10: Correlation execution data, target 10^{-5} 52

Table 11: Correlation execution data, target 10^{-10} 53

Table 12: All optimal solutions for the correlation benchmark 57

List of figures

Figure 1: Discrete precision approximation in a time interval 5

Figure 2: The complete coverage of the framework (Malossi, et al., 2018) 6

Figure 3: FlexFloat internal representation of a floating-point type 7

Figure 4: Pool-based active learning cycle 10

Figure 5: Visualization of the length limits of the mantissa on a 64-bits long backend 17

Figure 6: Regressor network for a 4 variables configuration 20

Figure 7: Variables Graph visualization of the correlation benchmark 23

Figure 8: General control flow of the script 26

Figure 9: Training set creation flow diagram 27

Figure 10: Optimization model solving algorithm, phase one 29

Figure 11: Optimization model solving algorithm, phase two (two versions) 30

Figure 12: Distribution for sampling the deviation of a single value in a configuration 34

Figure 13: Error and predicted error trends for each execution. Green dots correspond to the best solutions, yellow dots are generated solutions. 44

Figure 14: Bits sum trends for each execution. The red line identifies the sum of the optimal solution, while the green dots identify the sums for each best solution 45

Figure 15: Regressor metrics (left), Classifier metrics (right) (1) 50

Figure 16: Regressor metrics (left), Classifier metrics (right) (2) 51

Figure 17: Regressor metrics (left), Classifier metrics (right) (3) 52

Figure 18: Regressor metrics (left), Classifier metrics (right) (4) 53

1 Introduction

The energy consumption of computing systems is constantly growing and is a prevalent issue in a world tht increasingly values energy efficiency and sustainability. In this context, approximate computing techniques have been proposed in a variety of domains to design, both hardware and software, systems capable of making a compromise between the quality of the computed results and energy efficiency by which these results are produced. Among these strategies particular attention has been paid to precision scaling, a methodology which consists of changing the bit-width of program data to reduce storage and/or computing requirements (Tagliavini, Marongiu, & Benini, 2018).

The objective of this paper is to be a chapter in the development process of such techniques by joining an European project known as OPRECOMP and, more precisely, by becoming an integral part of the experimentations carried out by the Department of Computer Science and Engineering (DISI) of the University of Bologna in the purview of the application of AI and optimization techniques to regulate the energetic efficiency of high precision computations. In particular, this paper explores the possibility of applying a hybrid approach between Active Learning and Mathematical Programming to Transprecision Computing. This would entail embedding a machine learning model trained by means of an Active Learning approach into an optimization model to automatically and intelligently tweak the representation of floating-point numerical data. This project aims to lower the energetic expenditure of every single intermediate computation in a given program, while also avoiding errors that are systematically introduced when manipulating variables using this technique, and ensure that they do not exceed a maximum acceptable error rate decided prior.

This report is structured in three parts, spanning from the base concepts to the actual experiments and results. The first part (chapters 2, 3, and 4) covers the concepts of Transprecision Computing, Active Learning, and Mathematical Programming, while also presenting OPRECOMP, its goals and how they are related to this thesis. The libraries and tools used to develop and test this project are also presented here, with particular

attention on how FlexFloat (the Transprecision library) and EMLlib (a library to embed Ml models into optimization models) work internally, to explain and justify some decisions that were made during development.

The second part (chapter 5) analyzes and explores in detail the structure of the execution flow of the Python script produced for the experiments that will be mentioned in the final part. It first introduces the optimization problem and the model that was built to solve it and gives a full overview of the Machine Learning models, their structure and the reason why Active Learning proved to be essential to achieve good results. The paper then introduces the Variables Graph, a tool used to inject some knowledge about the relationships between variables of a program used to guide the optimization model to better solutions. How this tool works and what it tries to achieve are explained, and its limitations and potential issues are highlighted. The benchmarks used to execute the tests are also presented here, together with the reasons why some of the available benchmarks were discarded in favor of others.

Lastly, the second part covers in detail the execution flow of the script run during the experiments in a top-down manner, starting from the most general representation and then specifically addressing each phase of the algorithm in more detail, using a visual flow graph to aid the explanation.

The third part reports the most significant or interesting experiments and tests executed during the research. Other examples can be found in the Appendices, but the major considerations reported in this paper will primarily refer to the data in 6.

2 Transprecision Computing

In the purview of scientific computation, the *precision* of a value denotes the error which is introduced when a series of numerical data undergoes one or more mathematical operations in order to synthesize a new value. It is a clear indication of how bad a floating-point result can be in relation to the ideal result for those operations, thus giving a quantitative view of the reliability of the obtained value. In simpler words, lower is this error, higher is the precision.

Although high precision is preferable, trying to achieve the highest possible can prove to be sub-optimal. Let’s consider the human brain as an example: when it receives stimuli it’s not always the precision of the response that matters but rather, in some cases, how fast the response is computed (e.g. in a situation perceived as dangerous) or how much energy has been spent to compute it. Our brain can adjust the precision of its calculations to better fit the situation at hand. The same concept can be applied to automated computing by implementing a way to individually set the precision of variables and results depending on the domain, time limits and power efficiency requirements of the operation.

2.1 Concept

Traditional computing systems give developers and researchers the ability to decide what precision to give to variables used and stored during any computation. However, such systems take an ultra-conservative viewpoint, assuming that all floating-point calculations can be stored or performed with limited precision choices (e.g. 32 or 64 bits for single and double precision respectively or even 16 bits on newer platforms) provided by target platforms and enforcing the most conservative approach when inferring the precision of a computation result[1]. Although being a perfectly legitimate assumption, it

[1] For instance, when executing an arithmetic operation between a 32-bits long variable and a 64-bits long variable usual practice is, if not explicitly indicated, to return the result as a 64-bits long value.

has nothing to do with the actual application requirements and may become an obstacle when trying to lower cost and improve efficiency. This design principle has so far held thanks to the safe path to computational efficiency assured by Moore's law and its constant exponential improvement. However, the proliferate number of computational nodes, spanning from Internet of Things (IoT) nodes to High-Performance Computing (HPC) centers, and teir increasingly diverse power demands ranging from mWs (milli-Watts) or less to MWs (mega-Watts) mean that effective energy efficiency measures have become more and more essential (OPRECOMP, 2019).

Many floating-point operations and their related memory transfers emerge as the main bottleneck for energy efficiency, draining up to 50% of the overall system energy requirements (Mach, Rossi, Tagliavini, Marongiu, & Benini, 2018). To overcome this power wall, it becomes necessary to demolish such an old and conservative "precise" computing abstraction, replacing it with a more flexible and efficient alternative, namely Transprecision Computing, in which rather than tolerating errors implied by imprecise hardware or software components, systems are explicitly designed to deliver "just enough quality" (Tagliavini, Marongiu, & Benini, 2018).

Transprecision Computing is undoubtedly tied to the research area known as approximate computing but it also goes beyond the state-of-the-art and evolves it, as it controls approximation both at a finer grain (see Figure 1) and doesn't imply reduced precision at application level, despite allowing the exploitation and softening of application-level precision requirements for extra benefits (Malossi, et al., 2018).

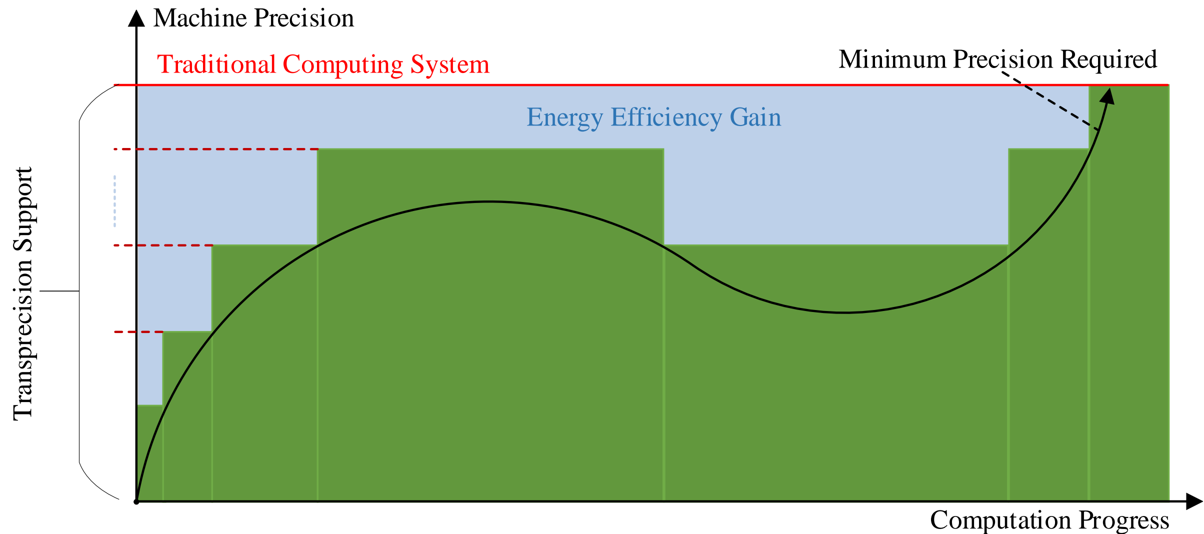


Figure 1: Discrete precision approximation in a time interval

2.2 OPRECOMP

Open Transprecision Computing (OPRECOMP) is a 4-years research project funded under the EU Framework Horizon 2020, coordinated by IBM and backed by a consortium of 10 among universities and companies that hold different key roles and focus on different tasks (OPRECOMP, 2019). The project goal is to build the foundation for computing based on transprecision analytics. The driving principle behind the approach is that almost any application involves a large number of intermediate calculations, whose accuracy is irrelevant to the final user, who is interested only in the reliability and validity of the final result. OPRECOMP wants to provide full support to Transprecision Computing through the development of a *transprecision framework* spanning all layers of computing systems, from devices, architecture and circuits design to software environment, tools, algorithms and data structures, along with the mathematical theory and physical foundations of the ideas (OPRECOMP, 2019).

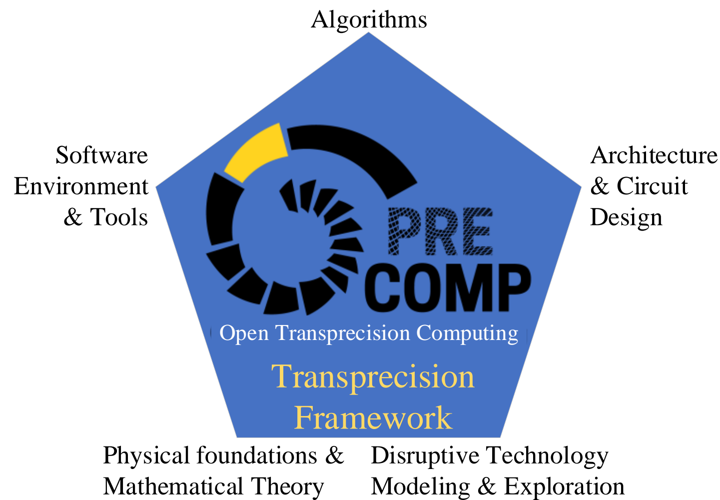


Figure 2: The complete coverage of the framework *(Malossi, et al., 2018)*

OPRECOMP is ultimately oriented toward developing and affirming a radically new computing paradigm. An ecosystem inspired by nature and human intelligence, designed to spend just the right amount of energy required for performing any particular operation. While high precision arithmetic will still be mandatory for financial applications, others such as data-mining or human-consuming applications will have the ability to adopt – or in some cases even embrace – less precise arithmetic despite the errors produced by underpowered or new emerging circuits technologies (OPRECOMP, 2019).

2.3 FlexFloat

There are many state-of-the-art tools and libraries available to developers and researchers which allow emulating arbitrary floating-point types of which MPFR (The GNU MPFR Library, 2019) or SoftFloat (Hauser, 2019) are among the most notables. While coupling floating-point emulation with precision tuning is the right approach to target both precision reduction in applications being developed for existing hardware (and thus IEEE floating-point formats compliant) and the specification of formats for new hardware being designed, these libraries are not designed for this purpose (Tagliavini, Marongiu, & Benini, 2018).

FlexFloat is an open-source software library[2] specifically designed to aid and improve the development of transprecision applications. Unlike those abovementioned, the FlexFloat library:

1. Uses an emulation methodology that leverages native host platform types to significantly reduce the time required to emulate custom types;
2. Enables accurate emulation of formats with arbitrary bit-width of mantissa and exponent fields;
3. Requires no modification at all to support custom types;
4. Provides advanced statistics on program variables that can be used by optimization models to enhance the solution and/or decrease the search time (Tagliavini, Marongiu, & Benini, 2018).

Since FlexFloat is not the main focus of this paper, the technical details regarding internal implementations of custom types and other features will be omitted from this paper. However, in order to better understand the objectives of what is going to be covered in chapter 5, it is important to clarify how this library is structured to represent floating-point data types and operations. FlexFloat internally represents floating-point types as a data structure composed of two unsigned integers (e, m) and a floating-point field (v) so that $e + m = len(v) - 1$, where *len(v)* is expressed in bits. v is also known as *backend value* and it's the system standard memory block that actually stores the value. The format of a type expressed by this data structure follows the conventions of the IEEE standard: 1 bit for the sign, e bits for the exponent and m bits for the mantissa. Its value can be computed using the formula $-1^{b_{m+e}} \times 0.(b_{m-1} \dots b_0)_2 \times 2^{(b_{m+e} \dots b_m)_2 - bias}$, where $bias = 2^{e-1} - 1$ (Tagliavini, Marongiu, & Benini, 2018).

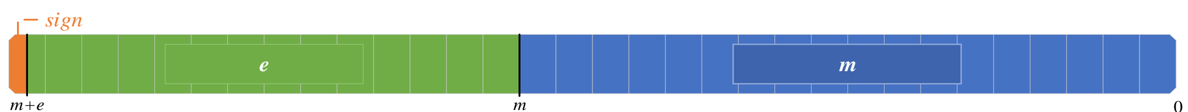


Figure 3: FlexFloat internal representation of a floating-point type

[2] Available at this link: https://github.com/oprecomp/

It’s also important to know that exactly like in classic computation operations between different precision variables cannot be executed without a cast. In FlexFloat this cast is considered exactly like another temporary variable that will store a value just for the duration of the computation and it is part of all the variables that are considered when fine-tuning the precision of intermediate computations:

```
1. var1 = var2 + var3 --> var1 = var4(var2) + var4(var3)
```

In this example, another temporary variable (`var4`) is added to the computation involving three variables. Just like the other three, `var4`’s precision can be tweaked before compile time.

In order to test the capabilities of this library, ten programs from different application domains were selected, constituting a benchmark suite that has been used (partially) to also test the performance of this project. Since all floating-point variables in the benchmarks are implemented as double types, all experiments presented in this paper have used a FlexFloat testing unit with a 64-bits backend (using standard hardware architectures).

3 Active Learning

When it comes to using a machine learning approach to a problem there are two initial issues to address:

- Acquire enough useful data for both training and testing;
- The type of learning task to use, chosen among supervised, unsupervised or a hybrid between the two (reinforcement learning won't be considered in this paper).

There are cases where it is really easy to both obtain and label enough data to build robust training and test sets and therefore where a supervised learning approach works best. There are also opposite cases where labeling data turns out to be so difficult, expensive or time-consuming that a supervised approach would just fail. Of course, it is always possible to use an unsupervised learning approach but, depending on the problem, it may not be a useful path to follow. It is in these last cases that Active Learning (or "Query Learning", in short AL) has empirically proven to be effective, although not without drawbacks.

3.1 Concept

The key hypothesis in Active Learning is that if the learning algorithm is allowed to choose the data from which it learns it will perform better with less training. It is a form of supervised learning applied to those cases where there are difficulties in acquiring labeled data: Active learning systems attempt to overcome this labeling bottleneck by asking queries in the form of unlabeled instances to be labeled by an *oracle*, usually a human annotator or an automatic tool (Settles, 2009).

Let's consider a classification problem with a target dataset $\mathcal{Z} = \{(x_1, y_1), \dots, (x_N, y_N)\}$ where x_i is a D-dimensional feature vector and y_i is its discrete-

valued label (that could be unknown), the standard Active Learning procedure (also known as a *pool-based Active Learning*) can be unfolded as follows:

1. The algorithm starts with a small labeled training dataset $\mathcal{L} \subset \mathcal{Z}$ and a large pool of unlabeled data $\mathcal{U} = \mathcal{Z} - \mathcal{L}$;
2. A classifier is trained using $\mathcal{L}$;
3. A query selection procedure picks an instance $x^* \in \mathcal{U}$ to be labeled;
4. x^* is given a label y^* by an oracle and both $\mathcal{U}$ and $\mathcal{L}$ are updated;
5. Repeat from (2) until the desired accuracy is achieved or the number of iterations has reached a predefined limit (Konyushkova & Sznitman, 2017).

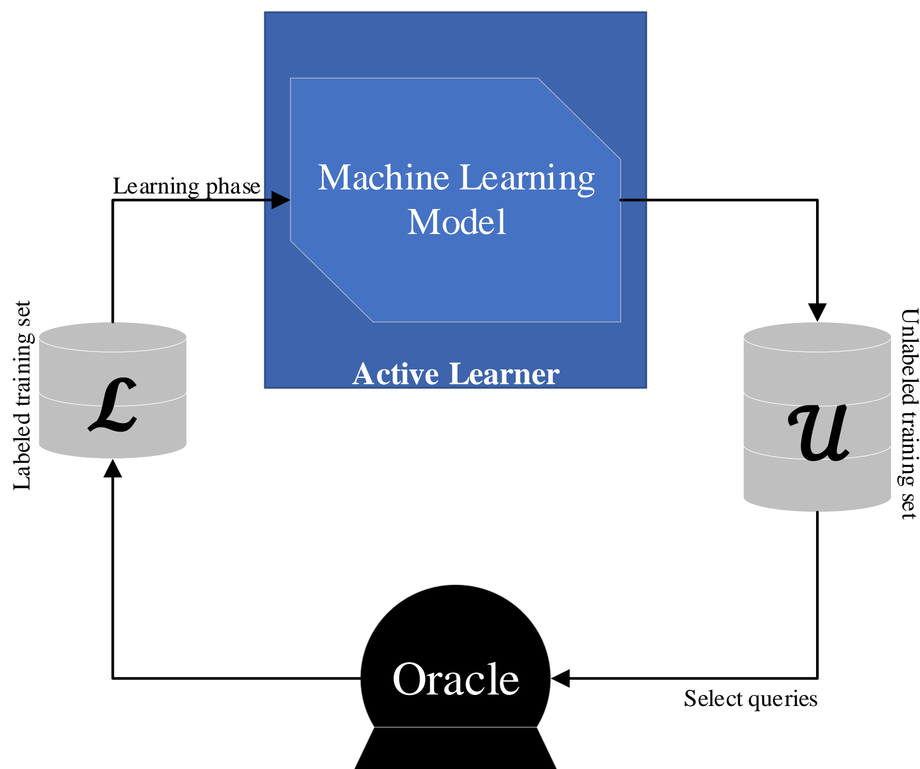


Figure 4: Pool-based active learning cycle

The pool-based Active Learning is the most commonly used sampling method, applied to fields like text, image or video classification, speech recognition, and information extraction, but there are other methods worth mentioning:

- Membership Query Synthesis, where the learner may request labels for any unlabeled instance in the input space. In this setting, the active learner generates synthetic instances to be labeled instead of selecting among a pool of samples. This method has been proven to reduce the predictive error rate more quickly than a comparable pool-based sampling. There is the possibility that the generated queries may be humanly unintelligible, making this approach inapplicable whenever the oracle is a human annotator (e.g. queries generating an artificial hybrid word with no meaning when attempting to recognize handwritten words).
- Stream-Based Selective Sampling, a method generally used when obtaining unlabeled instances is free or relatively cheap. In this case, the query can be sampled from the actual distribution and then the learner can decide whether or not to request its label (Kyu Hyun, 2017).

Once the sampling method has been established, the next step is to find a heuristic to select (or generate) the most informative unlabeled instances. There are several learning criteria that have been studied in literature, where the most noticeable are:

- Uncertainty Sampling, where the active learner will select a data point among those it is the least certain about (i.e. in a binary classification problem the one closest to 0.5 confidence);
- Query by Committee Algorithm, where a "committee" of machine learning models trained with the same label instance is established. The task of each model of the committee is to vote on labeling unlabeled instances, and the one the committee disagree the most about is selected as the next query.

These two criteria, although commonly used in many Active Learning Applications, haven't been applied in this specific instance and therefore won't be discussed further. For a more detailed explanation refer to *"Explaining Active Learning Queries"* by Kyn Hyun Chang, 2017.

4 Mathematical Programming

Also known as *Mathematical Optimization*, Mathematical Programming (in short MP) is – in its broadest sense – the selection of the best element from a set of alternatives, following a certain criterion. In the most common case, an optimization problem consists in minimizing or maximizing a target function by assigning values to input variables from an allowed set which gives the best result with regard to the target function. A more rigorous definition can be expressed as follows:

Let $F \subseteq \mathbb{R}^n$ be the set of feasible solutions (corresponding to all vectors x satisfying the constraints of the problem), and $d: F \to \mathbb{R}$ a function associating a real value to each feasible point of F. The couple (F, d) defines the optimization problem that consists of finding a point $f \in F$ (global optimum) such that $d(f) \leq d(y)\ \forall y \in F$.

(Martello, 2014)

The short notation to express an optimization problem given a target function d is

$$\min_{x \in F} d(x)$$

or

$$\arg\min_{x \in F} d(x)$$

if the focus is the value of x that minimizes d rather than $d(x)$ itself.

MP is a wide field of study spanning many different approaches for different scenarios, such as Convex Programming (in turn, a broader category which includes Linear programming), Integer Programming, Non-linear Programming, Combinatorial Optimization, Constraint Optimization, and many others. In the scenario examined by this paper, the nature of the decision variables (discrete-valued) and constraints lead to the use of Mixed Integer Programming techniques. These techniques involve using linear or quadratic programming relaxations to compute bounds on the value of the optimal solution. They also use linear programming and other techniques to compute linear

constraints that cut off possible solutions that violate the discreteness constraints (IBM, 2019).

4.1 CPLEX and DOcplex

CPLEX is a high-performance mathematical programming solver developed by IBM. Originally the first linear optimizer commercially available written in C (its name comes from the union of C and simplex), CPLEX has evolved over timeand now implements mixed integer programming and quadratic programming (IBM, 2019).

Part of the CPLEX environment is the Decision Optimization CPLEX Modeling library for Python (in short DOcplex), a wrapper for the C solver callable using the Python API. It allows solving optimization problems on Cloud service or on the local machine. Fully configurable, it is composed of 2 modules:

- Mathematical Programming Modeling for Python using `docplex.mp`[3] (DOcplex.MP);
- Constraint Programming Modeling for Python using `docplex.cp`[4] (DOcplex.CP) (IBM, 2019).

4.2 Empirical Model Learning

Empirical Model Learning (in short EML) is a technique to enable Combinatorial Optimization and decision making over complex real-world systems. The method is based on the principle of using a Machine Learning model to approximate the behavior of a system that is hard to model by conventional means. The result is an Empirical Model (hence the name) to embed into a Combinatorial Optimization model (Milano & Lombardi, Empirical Model Learning | Embedding Machine Learning Models in Optimization, 2019).

[3] The documentation of this module is available at https://ibmdecisionoptimization.github.io/docplex-doc/mp/refman.html
[4] The documentation of this module is available at https://ibmdecisionoptimization.github.io/docplex-doc/cp/refman.html

In EML the problems of interests are usually defined over high-complexity systems, typically structured as follows:

$$\min f(x, z)$$

so that

$$g_j(x, z) \qquad \forall j \in J,$$
$$z = h(x),$$
$$x_i \in D \qquad \forall x_i \in x$$

where x is a vector of decision variables x_i with domain D_i (with no special assumptions on D_i) and z is a vector of observables related to the target system. $g_j(x, z)$ represents a series of logical predicates corresponding to classical inequalities from Mathematical Programming or to combinatorial restrictions. The $h(x)$ function models the complex behavior of the system and specifies how the observables z depend on the decision variables: it corresponds to the encoding of the Empirical Model obtained via Machine Learning (Milano, Lombardi, & Bartolini, Empirical Decision Model Learning, 2017).

Embedding an Empirical Model into an optimization model requires encoding the Empirical Model in terms of variable and constraints and therefore exploit the underlying optimization approach to boost the search progress, e.g. via bound computation or constraint propagation. Although the EML Python library[5] (developed by the University of Bologna) had a major role in the development of this project it will not be discussed further, and its technical details are left to the reader to explore.

[5] Available at this link: https://github.com/emlopt/emllib

5 Project and objectives

This paper will cover some experiments conducted by the University of Bologna as part of the OPRECOMP research project. These experiments aim to develop a reliable decision-making system in the field of transprecision computing by applying a hybrid approach between active learning and mathematical programming. The main goal is to achieve comparable results to those achieved by state-of-the-art tools and methodologies while drastically reducing the execution time. This paper covers the results and experiments conducted by means of a Python script executed locally derived from the work by Borghesi Andrea.

A program using n FlexFloat variables (also counting the temporary ones) requires that each of these variable be set to the lowest precision possible given a maximum tolerated error e on the final result. As stated in chapter 2.3, the backend of each variable is a native *double* type and thus a 64-bit long value. Considering 1 bit for the sign, there are 63 bits to assign to either the exponent or the mantissa. It has been arbitrarily decided to leave a limit of minimum 10 bits for the exponent and 4 bits for the mantissa, thus effectively limiting the length of the latter to the interval [4, 53].

Figure 5: Visualization of the length limits of the mantissa on a 64-bits long backend

The target decision-making system being designed is composed of two models: an optimization model functioning as the main solver and a machine learning model predicting the error on the final result of the system undergoing study, embedded into the optimization model via the EML library.

5.1 Optimization Model

In order to enforce a proportional relationship between precision and results, the mantissa's bit-lengths (m in FlexFloat's floating-point data type) were chosen as decision variables in the optimization model. Let's then consider the vector c of all m_i values of each ith variable in a target program (from now on this vector will be referred to as *configuration*). We can express the model as follows:

$$\text{argmin} \sum_{i=1\ldots n} c_i$$

with

- $z = h(c)$, where $h(c)$ is the active learning model
- $c_i \in [4 \ldots 53]$, $i = 1 \ldots n$
- $z \leq e$, where e is the target error.

It's important to specify that generally $0 < e \ll 1$ and as such it tends to assume values that are difficult to use for exact comparisons on a machine. To account for this issue, the model has been modified to this formula:

$$\text{argmin} \sum_{i=1\ldots n} c_i$$

with

- $z = h(c)$, where $h(c)$ is the machine learning model
- $c_i \in [4 \ldots 53]$, $i = 1 \ldots n$
- $z \geq -\log(e)$, where e is the target error.

The choice was guided by the fact that errors expressed as negative orders of magnitude $10^{-\mathrm{m}}$ can also be expressed as $-\log(10^{-\mathrm{m}}) = m$. This new formulation inverts the error constraint, since higher m values correspond to more negative orders of magnitude and thus to smaller errors, but also widens the interval of the error value ensuring both better precision on the comparisons and a more ergonomic way to pass the target error as a parameter.

This base model discussed up to this point, although correct, has been expanded and refined during development to overcome some implementation and performance issues that arose while undergoing testing (see sub-chapters 5.2, 5.3 and 5.4).

5.2 Machine Learning Model and Active Learning

The function binding variable precision and final error is complex and impossible to express analytically. This is why it was impractical to try expressing it through a combination of constraints and expressions directly in the optimization model and why it has been chosen to instead express them through a Machine Learning model. In this scenario, EML turned out to be necessary for successfully blend both the optimization and the machine learning models together and allow the usage of the best of both worlds.

Although the obvious requirement of using a regressor to predict the error (in logarithmic form) given a configuration, the Active Learning phase actually involves the usage of two different Machine Learning models: a neural network used as a regressor and a decision tree used as a binary classifier. The first is used as intended and is embedded into the optimization model as the h function, while the latter is used as an additional tool to discard widely inaccurate solutions. It is also embedded into the optimization model as well and forces it to discard all solutions which it classifies as configurations with an error higher than a certain threshold, hence giving to the optimization model some knowledge about how variables impact the final error. The optimization model with the addition of the classifier can be expressed as the following:

$$\text{argmin} \sum_{i=1 \dots n} c_i$$

with

- $z_r = h_r(c)$, where $h_r(c)$ is the machine learning model of the regressor
- $z_c = h_c(c)$, where $h_c(c)$ is the machine learning model of the classifier
- $c_i \in [4 \dots 53]$, $i = 1 \dots n$
- $z_r \geq -\log(e)$, where e is the target error
- $z_c < 0.5$.

The structure of the regressor depends on the number of variables in the program and its input is the configuration. It is composed of an input layer, two dense hidden layers and a fully connected output layer composed of a single neuron with a linear activation function. The first hidden layer is composed of twice the number of features (which at the time of writing this paper corresponds to the number of variables n) neurons while the second is composed of exactly n neurons. Both the hidden layers use a rectifier[6] activation function.

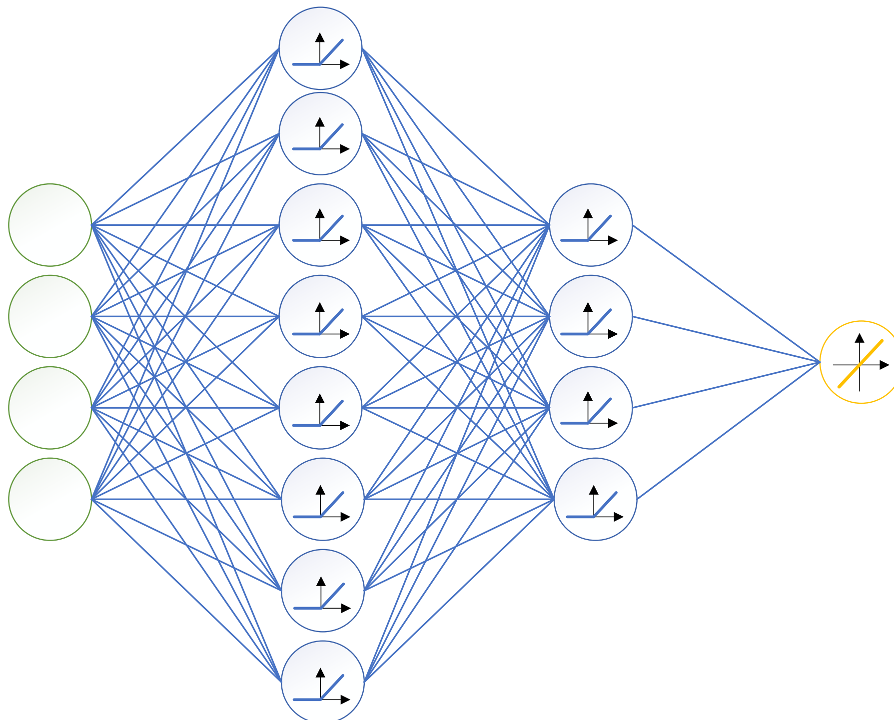


Figure 6: Regressor network for a 4 variables configuration

The structure changes with programs that have high number of variables, in which case the first hidden layer is divided into two distinct fully-connected layers with $n/2$ and $n/4$ neurons repectively. Both the architectures are the result of empirical evaluations, but it's extremely probable they are not the best choice available. There are studies being conducted to determine if AutoML techniques[7] could be applied to this problem to improve the network architecture.

---

[6] A REctified Linear Unit (relu) is a unit using the rectifier function $f(x) = \max(0, x) = x^+$.
[7] AutoML stands for automated machine learning and refers to a series of techniques to automate the whole machine learning pipeline, from data manipulation to the training of the model (Xin, Zhao, & Chu, 2019).

Given a program of n variables, there are $(53-3)^n = 50^n$ different possible configurations. It means that to build a training set big enough to obtain a reliable Machine Learning model requires an exponentially increasing time with n. The Active Learning approach aims to achieve comparable results by starting with a very limited number of training entries (1000 in each experiment) and refining the ML model with the aid of the optimization model. The refinement process can be described as a variation of a Membership Query Synthesis with an Uncertainty Sampling heuristic naturally obtained through the relation between the regressor and the optimization model: when tasked to find a solution, the MP model uses the regressor to synthesize a configuration, which is then used to run the program and verify its error. If the solution returned by the MP model happens to be unfeasible it means that the regressor made a wrong prediction and thus that specific configuration is one the regressor is uncertain about. Using it to retrain the regressor could potentially improve its accuracy for the next attempt.

5.3 Benchmarks and Variables Graph

The same benchmark suite build for FlexFloat was used to test the system. The suite's programs represent common algebraical and mathematical operations widely used in the world of scientific and engineering computations. Some of them couldn't be used for lack of data and the experiments reported on this paper refer solely to the correlation and convolution operation benchmarks. Before moving on it is important to note that the correlation benchmark, at the time of gathering the data reported in this paper, suffered a malfunction which has been discovered only during later stages of the experiments. The wrong values returned by the benchmark don't void the data presented here per se but render them incomparable to data coming from a functioning version of the same benchmark, without nullifying the observations and comments in the following chapter.

To accelerate the entire process, a tool was introduced to inject into the MP model some knowledge about the variables of the benchmarks and their relations. For instance, it is legitimate to think that if a variable V is assigned to another variable U, they both should have the same precision or U should preferably be more precise than V, just like it is preferable to assign a *float* variable to a *double* and not vice-versa. This tool gathers

all the information regarding how the variables interact with each other in a graph data structure called Variables Graph. It stores relations between variables in case of:

- Two variables that are part of an assignment;
- Two variables that are part of the same expression;
- Two variables that are linked by a cast operation (the case of temporary variables);
- A variable that is the formal parameter of a function and the other is the corresponding actual parameter.

From a Variables Graph it is possible to extract some semantic information that helps the MP model prune many configurations from the feasible set. For instance, let's consider the following assignments:

```
1. var1 = var2 + var3
2. var5 = var6
```

As already mentioned in chapter 2.3, they are transformed by FlexFloat into this form:

```
1. var1 = var4(var2) + var4(var3)
2. var5 = var6
```

Following the convention of incrementally naming all variables, let's rename variables as 'T' if they're temporary or 'V' otherwise.

```
1. V1 = T4(V2) + T4(V3)
2. V5 = V6
```

For both the assignments, if the right-hand side expressions (respectively the variables T4 and V6) aren't used anywhere else, it would be wasteful to allocate to them more memory than the variable their value is going to ultimately be stored into. Therefore, it can be derived that $\text{len}(T4) \leq \text{len}(V1)$ and $\text{len}(V6) \leq \text{len}(V5)$. Furthermore, the first expression casts two variables with (potentially) different precision to a temporary variable $T4$: since FlexFloat makes the design choice of setting the precision of temporary variables to the minimum of the operands, the same relation $\text{len}(T4) = \min\big(\text{len}(V2), \text{len}(V3)\big)$ can be enforced as a constraint in the MP model.

Although useful for limiting the solutions pool size, the Variables Graph may impose constraints that are too strict and end up pruning the optimal solution. Here’s an example using the correlation benchmark:

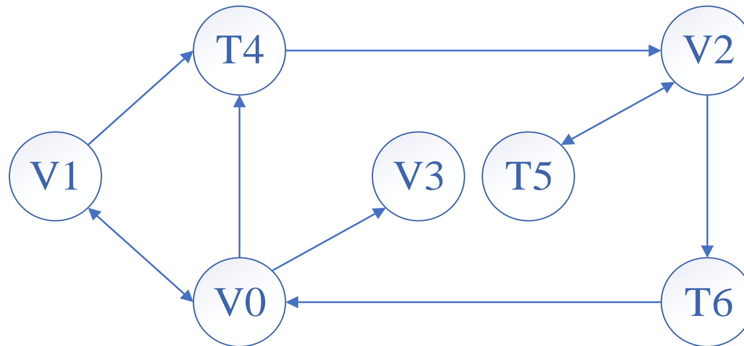


Figure 7: Variables Graph visualization of the correlation benchmark

Figure 7 shows the Variables Graph of the correlation benchmark. Each node represents a variable (names don’t reflect those actually used in the source code of the benchmark) and each arrow represents the relationship between the variables it is connecting. Each A → B relationship corresponds to an expression that is traceable to

```
1. B = A
```

hence to the relation $\text{len}(A) \leq \text{len}(B)$. The following can be derived from the graph in Figure 7:

- $\text{len}(V0) \leq \text{len}(V1)$ and $\text{len}(V1) \leq \text{len}(V0) \rightarrow \text{len}(V0) = \text{len}(V1)$;
- $\text{len}(V0) \leq \text{len}(V3)$;
- $\text{len}(V0) \leq \text{len}(T4)$;
- $\text{len}(V1) \leq \text{len}(T4)$;
- $\text{len}(V2) \leq \text{len}(T5)$;
- $\text{len}(V2) \leq \text{len}(T6)$;
- $\text{len}(T4) \leq \text{len}(V2)$;
- $\text{len}(T5) \leq \text{len}(V2)$;
- $\text{len}(T6) \leq \text{len}(V0)$.

Moreover, the length of a temporary variable (a 'T' node) must be equal to the minimum length of all its predecessor nodes (variables):

- $\text{len}(T4) = \min(\text{len}(V0), \text{len}(V1)) = \text{len}(V0) = \text{len}(V1)$;
- $\text{len}(T5) = \text{len}(V2)$;
- $\text{len}(T6) = \text{len}(V2)$.

Given a generic configuration $[V1, V2, V3, T4, T5, T6]$ as the solution of the optimization model, these constraints limit it to all configurations of the form $[a, a, a, b, a, a, a]$ with $a \leq b$, constituting about the 0.000016% of the total number of possible configurations. Nevertheless, despite the substantial pruning achieved and the consequent time saving, the optimal solution is also excluded. For instance, the optimal solution for a target error of 10^{-5} found using state-of-the-art tools is $[13, 4, 13, 13, 11, 12, 11]$ which clearly does not respect the imposed constraints. Another example is found in Appendix I.

With the integration of the Variable Graph-derived constraints, the optimization model can be formalized as follows:

$$\text{argmin} \sum_{i=1\ldots n} c_i$$

with

- $z_r = h_r(c)$, where $h_r(c)$ is the machine learning model of the regressor
- $z_c = h_c(c)$, where $h_c(c)$ is the machine learning model of the classifier
- $c_i \in [4 \ldots 53]$, $i = 1 \ldots n$
- $c_j \leq c_k \; \forall (c_j, c_k) \in G$, where G is the set of all couples of connections $c_k \rightarrow c_j$ of the graph
- $c_j = c_k \; \forall c_j \in T$ and $c_k \in A_{c_j}$, where T is the set of all temporary variables and A_{c_j} is the set of all successors of c_j in the graph
- $z_r \geq -\log(e)$, where e is the target error
- $z_c < 0.5$.

5.4 Structure

The script used to run all the experiments discussed in this paper is a refactored and modified version of the original script. The script has been re-organized and modularized to make it more readable and extendable, in addition to a more complete arguments system to easily tweak all the parameters used during the execution. The version of the project referred to in the next pages can be found in the Git repository at https://github.com/Acciai0/oprecomp_project, with a complete guide to run it.

The new architecture of the script is composed of the following main modules:

1. `argsmanaging`. This module deals with starting arguments and builds a global data structure from which to retrieve all initialization values when needed. It is structured so that it is easy to add new labeled parameters or remove unused ones. It automatically handles type conversions from the strings coming from console arguments and all errors that could occur meanwhile. All possible arguments are listed by executing `python3.al  -help` or in the already mentioned Git repository.
2. `benchmarks`. Module in charge of listing, loading and executing benchmarks. It is used by the `argsmanaging` module to load all data relative to the given benchmark, such as the construction of the Variables Graph, the number of variables and other information (see the `Benchmark` class).
3. `training`. This module manages the construction of a training session, i.e. of a training set and a test set, and the creation and training of both the regressor and the classifier by means of trainer objects specific for the type of model (neural network or decision tree, see `RegressorTrainer` and `ClassifierTrainer` classes and the respective sub-classes). It has been built to be easily expandable to also manage other types of regressors or classifiers if needed.
4. `optimization`. The core of the script. This module is in charge of building the optimization model and iterating until it is solved. It articulates the macro phase of execution described in Figure 8 and exposes functionalities to dump the entire execution to a log JSON file for external purposes (they've been used to generate the tables and figures in chapter 6).

5. `data_gen`. This module contains all functionalities used to manipulate examples for the Active Learning phase and for the Neighborhood Refinements, such as interpolating and generating routines.
6. `utils`. A utility module which aids the execution by providing utility functions such as I/O routines for reading/writing benchmark-specific files or printing utilities for formatted and colored output (`print_n`()). It also contains a stopwatch object used to time all critical operations in the program.

This section wants to better define the structure and the control flow of the script and its evolution, starting from the original version and exploring what has been changed in order to obtain the results reported here.

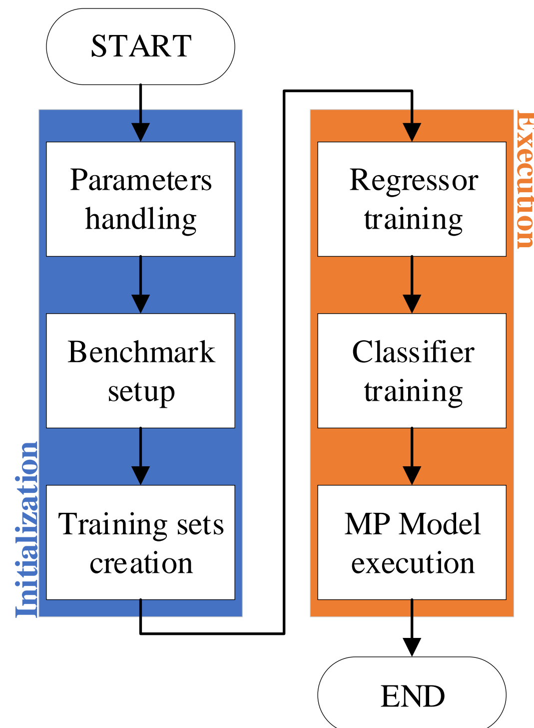


Figure 8: General control flow of the script

Figure 8 shows the general control flow of both the original and the refactored script: it can be decomposed into two macro phases of initialization and execution where the first phase groups all the operations needed to gather and initialize the required data, while the latter includes all Active Learning applications and optimization steps.

Parameters handling

Checking of the presence of all mandatory parameters and of their value. In the new script, parameters are labeled and used to tweak not only target error, type of regressor and classifier and the benchmark, but also other internal parameters like the changing rate in the neighbor search (see paragraphs further on) or the delta of orders of magnitude to limit the search into.

Benchmark setup

This phase gathers all data regarding the benchmark specified as a parameter, dynamically getting the total number of variables used in the program, its location on the disk and building its Variables Graph and therefore gathering all the relationships among the variables.

Training sets creation

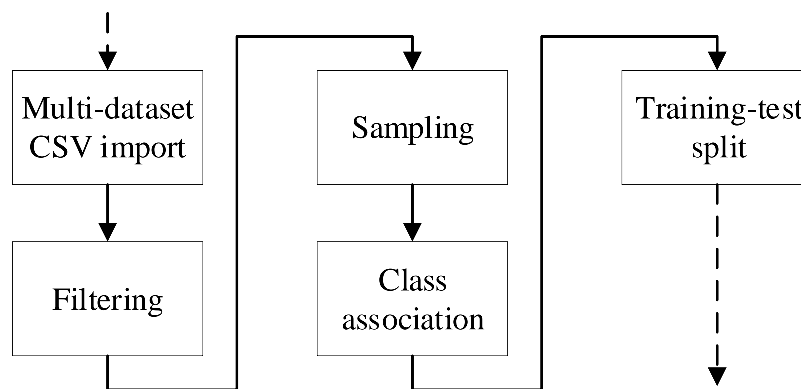


Figure 9: Training set creation flow diagram

The script works with a series of datasets built differently from one another. All results in this paper have been calculated on the basis of data coming from dataset number zero. First and foremost, the dataset is loaded from the respective CSV file, then filtered to remove any entry with all columns equal to zero. All errors higher than a certain threshold passed as a parameter are clamped to the value of that threshold and a new column containing the logarithm of the errors is also added to the dataset. At this point, the script

randomly samples half of the final number of entries from the dataset and the other half with the following distribution:

- 40% among entries with lower bits sum (these are more critical for training since the regressor has to be as precise as possible when predicting the lower end of the spectrum);
- 30% among entries with low bits sum;
- 20% among entries with high bits sum;
- 10% among entries with higher bits sum.

For instance, in the case of the correlation benchmark, 40% of the entries will have bits sum between 28 and 114, 30% between 115 and 200, 20% between 201 and 285 and 10% between 286 and 371.

The last steps are the addition of a column specifying the class of each configuration based on the error threshold mentioned prior (errors higher than the threshold are labeled 1, otherwise they're labeled 0) and the splitting of the resulting dataset into a training set and a test set. The ratio between the test set and the training set is specified via parameter.

Regressor and Classifier training

The training set built in the previous step is used to train both the regressor and the classifier, then they are both tested on the test set to compute some quality metrics (e.g. Mean Squared Error, Mean Absolute Error, Accuracy). The test set is kept until the end of the execution in order to recalculate the metrics after each iteration, while the training set is used again only to retrain the Decision Tree, which cannot be trained using an Active Learning approach.

MP model execution

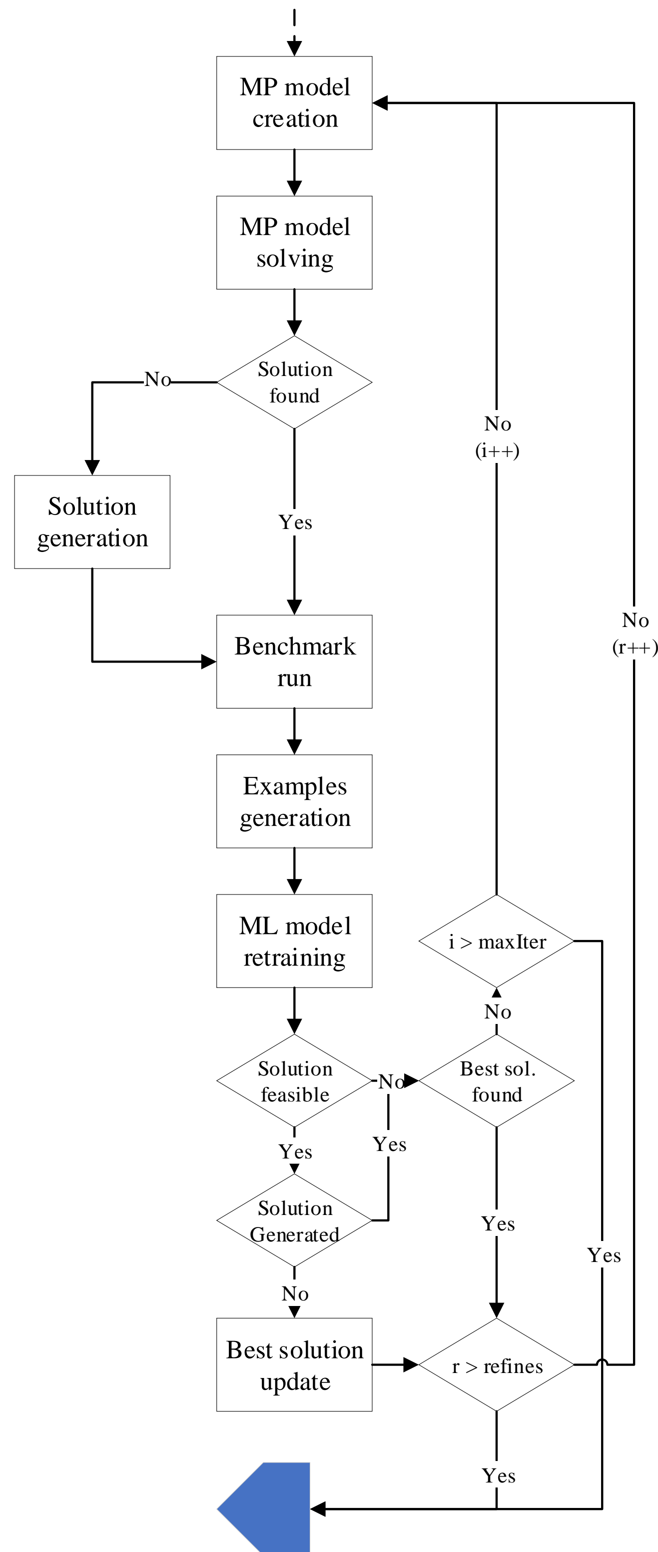


Figure 10: Optimization model solving algorithm, phase one

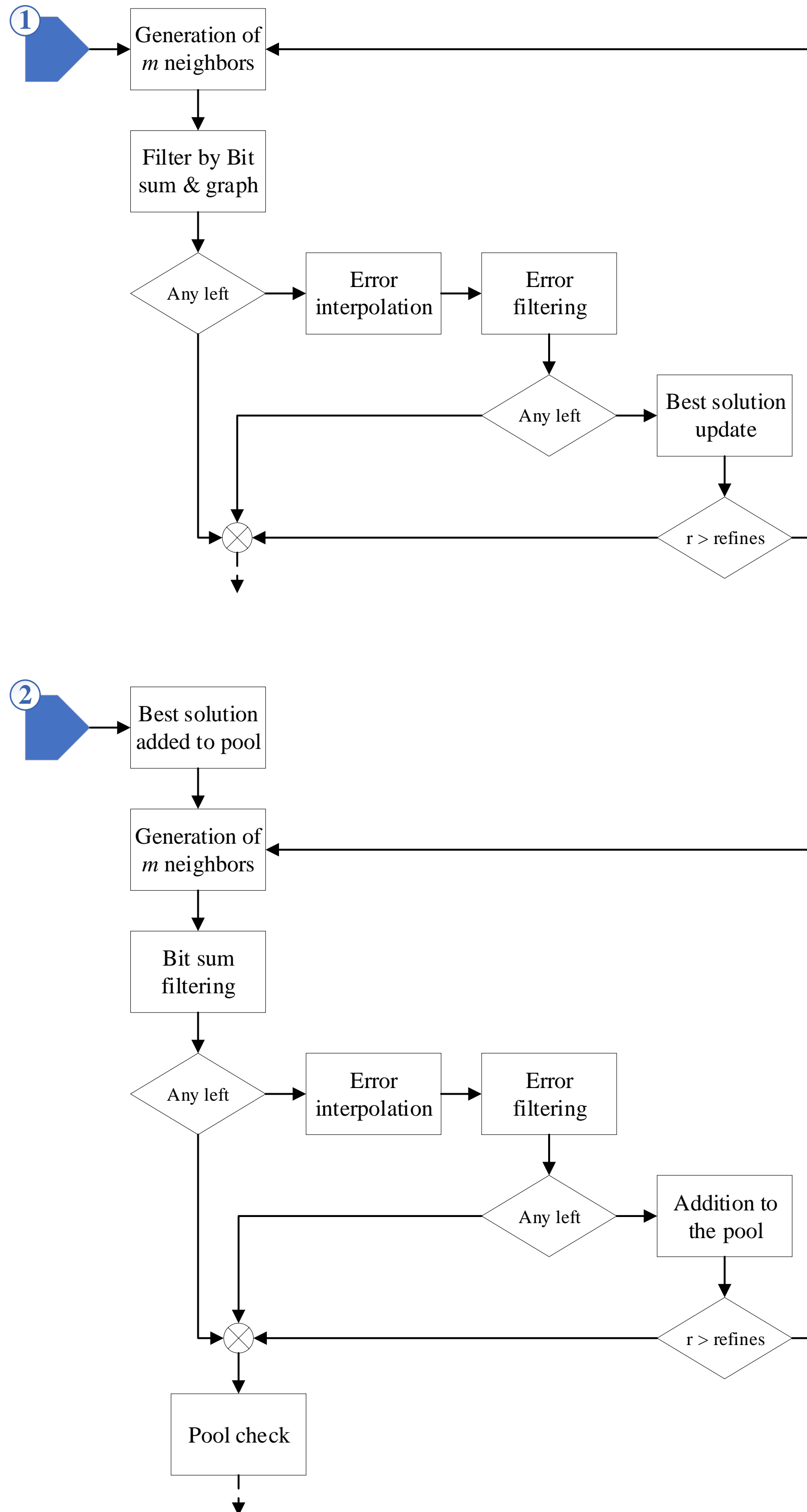


Figure 11: Optimization model solving algorithm, phase two (two versions)

This macro step is the core of the entire process and the most time-consuming. It is composed of two phases: the first phase of optimization model solving and the second one of local refinement. The first phase is a loop that iterates for a certain number of times or until a feasible solution has been found.

MP model creation

The first step of a single iteration in phase one is the creation of an optimization model. The reason why the same model cannot be used more than once depends on how the EML library embeds a ML model into the MP model. Since what the library does is essentially transforming the structure of a neural network (or decision tree) into the specific representation of the optimization model solver, any retraining carried out on the ML model won't be transmitted to its embedded representation.

At each iteration, all previous solutions deemed unfeasible are used to define a series of new constraints in order to remove them from the solutions pool and an additional constraint is added to force the solver to find a better solution than the last feasible one it has found (if any). This last constraint is only used for the additional iterations after the first feasible solution has been found, the number of which is specified through a parameter. The final optimization model has the following form:

$$\text{argmin} \sum_{i=1\dots n} c_i$$

with

- $z_r = h_r(c)$, where $h_r(c)$ is the machine learning model of the regressor
- $z_c = h_c(c)$, where $h_c(c)$ is the machine learning model of the classifier
- $c_i \in [4 \dots 53]$, $i = 1 \dots n$
- $c_j \leq c_k \ \forall (c_j, c_k) \in G$, where G is the set of all couples of connections $c_k \rightarrow c_j$ of the graph
- $c_j = c_k \ \forall c_j \in T$ and $c_k \in A_{c_j}$, where T is the set of all temporary variables and A_{c_j} is the set of all successors of c_j in the graph
- $\sum_{i=1\dots n} c_i \leq \sum_{i=1\dots n} c_i^s \ \ \forall s = 1..m-1$, m being the current iteration

- $z_r \geq -\log(e)$, where e is the target error
- $z_c < 0.5$.

MP model solving

Invocation of the solver and computation of a solution to the optimization problem. If no solution is found (mostly due to the insufficient or poor training of the regressor/classifier) the system generates a random configuration, which serves the purpose of giving some data to work with through the next steps, even without an actual solution. In case the configuration passed to the next steps was generated, it's not important if it was feasible or not since its role is just to be used to retrain the regressor and, to a lesser extent, the classifier.

Solution generation

If no solution is found during the optimization process (i.e. the solver has concluded that the problem cannot be solved) the most probable cause of the error is a bad training of the regressor, which is temporarily unable to predict low enough values, regardless of the configuration submitted to it. To overcome this error state the regressor needs to be retrained with new entries and because the solver can't provide a new one, a random configuration is generated. Initially, instead of generating a totally random configuration the script used the previously found solution (feasible or unfeasible) and increased by one unit each value. This choice worked well except if the previous solution was maximized, i.e. $[53, \dots, 53]$, in which case the new training would have had, in the best scenario, no effect at all.

To avoid this problem, the version used in this paper generates a completely random sequence clamped between $[4, \dots 4]$ and $[53, \dots, 53]$, which served its purpose, whilst acknowledging that the solution could be much improved in the future.

Benchmark run

Whether it is generated or not, the configuration calculated at the previous step is fed to the benchmark and the actual final error is calculated. This step is the most time-consuming, but also the most accurate and its result will be used to check if the solution is feasible or not and as an oracle to retrain the ML models, applying the Active Learning approach automatically.

Examples generation

A single example doesn't usually affect the training quality in any critical way, therefore it's important to synthesize a bigger batch of examples to feed to the ML models training session. Using a simple neighborhood search given all the possible values is not applicable in this case, due to the impracticable number of possibilities and thus the memory required (for a benchmark with just seven variables the machine would need $50^7 * 4 * 7$ bytes, which is about 20TB of data). So, to build a batch of examples, the script instead *generates* new entries using the following algorithm:

```
1. function GENERATE_NEIGHBOR(configuration,
   singleChangeProbability, min, max)
2.   n = EMPTY_LIST
3. for each value v in configuration
4.   p = random value between 0 and 1
5.   if p <= singleChangeProbability then
6.           delta = int(random sample from a normal distribution
   with μ=0 and σ=2)
7.           v = v + delta
8.           v = CLAMP(v, min, max)
9.   APPEND v to n
10.  return n
11.
12.  function GENERATE_NEIGHBORS(configuration,
   singleChangeProbability, min, max)
13.  neighbors = EMPTY_LIST
14.  neighborsCount = int(1/singleChangeProbability) * 10
15.  while not generated neighboursCount neighbors
16.          n = GENERATE_NEIGHBOR(configuration, min, max)
17.          APPEND n to neighbors
18.  return neighbors
```

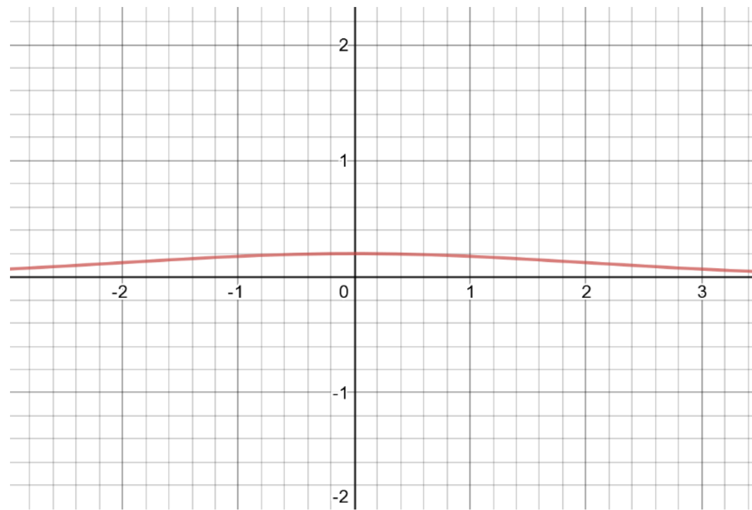


Figure 12: Distribution for sampling the deviation of a single value in a configuration

Of course, these entries also need to have an error associated with them. Three different methods have been examined for the task:

- Piecewise linear interpolation. This algorithm has been optimized for data lying on a structured grid while the configurations reported in any dataset are intrinsically arranged in an unstructured manner;
- Radial Base Function interpolation. A powerful mesh-free interpolation algorithm for high-dimensional data. It's been discarded in favor of the Nearest Neighbor interpolation algorithm.
- Nearest neighbor interpolation. A fast algorithm that assigns to configurations the same error of the nearest neighbor inside the search space. Assuming that among configurations of the same neighborhood the error doesn't vary too much, the lower precision than the RBF interpolation on the results can be accepted in exchange for a substantial increment of the execution speed.

ML model retraining

The last result and all the examples generated from it (removing duplicates) are used to train the regressor and then they are added to the original training set to retrain the classifier anew (it's a quick enough process that no incremental training is needed).

The next step is to check if the given solution is feasible or not: the optimization model returned a configuration on the basis of the regressor's prediction, which could be wrong, potentially by many orders of magnitudes. In this case, the solution is marked as unfeasible and discarded, while if it turns out to be feasible, it is saved as the current best solution and the algorithm starts another cycle comprising a few more iterations, trying to further improve the solution.

Whether it is because the algorithm cycled for the maximum number of times or it found a feasible solution and finished trying to improve it, in the end, the algorithm will enter the phase two, where it tries to refine the best solution found in the previous phase by executing a neighborhood search for a limited amount of times. This phase has been implemented in two different ways (Figure 11), both with pros and cons. What follows is how the first version works and how it differs from version two.

Generation of m neighbors

The neighborhood search starts by generating more neighbors in the same way as outlined previously, starting from the current best configuration.

Filter by Bit sum/graph

All configurations found in the previous step are filtered to retain only those with a lower bits sum than the current best and that satisfy the constraints given by the Variables Graph. If there isn't any configuration left after filtering, the loop breaks and the last best solution found is returned.

Error interpolation

To avoid long execution times, this step uses an interpolation method to assign to each configuration an error, instead of calculating it by executing the benchmark. This way the algorithm saves execution time, but it also risks labelling configurations as feasible although that aren't. To minimize the risk, instead of using the Nearest Neighbor interpolation it uses the Radial Base Function interpolation method.

Error filtering and Best solution update

All configurations with an error higher than the target are discarded and, among those left, the one with the lowest bits sum is chosen as the new best solution.

The second version of phase two works similarly to version one, with a couple of differences:

- Neighbors aren't filtered using the Variables Graph relationships;
- At each iteration, if a better configuration was found it is added to a pool of solutions;
- At the end of all iterations (or when no more feasible configurations are found), all the configurations in the solutions pool are tested by running the benchmark

from best to worst and the first solution found to be feasible is returned as the best solution. Compared to version one, this version is a lot more time-consuming, but it can potentially improve the solution found in phase one beyond the constraints enforced by the Variable Graph.

6 Results

This chapter reports the results of some experiments ran on different benchmarks and different target errors. It is structured in three parts, each of which focuses on a different part of the execution.

Each experiment will be reported using the following structure:

IT. #	CONFIGURATION	GENERATED	ERROR	PREDICTED
0	[…, …, …]		e_1	p_1
…	…	…	…	…
K	[…, …, …]	✔	e_k	p_k
…	…	…	…	…
N	[…, …, …]		e_n	p_n

Neighborhood refinements
—

Best solution […, …, …] found in ?s
Optimal solution: […, …, …]

Table 1: Experiment report structure

For each optimization iteration, it contains the respective configuration used as a possible solution, as well as an indication of if the given configuration was generated and its error and the prediction of its error coming from the regressor. If there was any configuration coming from a neighborhood refinement, it is listed here, together with the final result, the time spent to find it, and what it is the optimal result. The best solution will be reported at the bottom of the table and, for convenience, it is also highlighted in light blue in the corresponding iteration row. All errors lower than the target are highlighted the same way to help the reader to easily identify those configurations which have been proven to be

feasible. Predicted errors are not highlighted because the regressor, by construction, will always return feasible errors except for generated solutions, which are anyhow ignored as possible solutions. It's important to keep in mind that the best solution is not, in most cases, the one reported in the last iteration, because after the first solution is found (the first with a highlighted error) the script proceeds for a certain amount of extra iterations trying to find an even better solution. The number of extra iterations can be specified as program parameter and, by default, is set to 4.

6.1 Consistency Tests

The results reported in this section refer to those experiments carried out to test the variance of the results among different executions with the same parameters. The test can only be considered passed if the script can consistently generate comparable solutions at each execution. Here follows an example of five executions of the correlation benchmark, with target error of 10^{-7} and no neighborhood refinements.

IT. #	CONFIGURATION	GENERATED	ERROR	PREDICTED
0	[15, 15, 15, 34, 15, 15, 15]		1.324e-07	1.499e-12
1	[16, 16, 16, 16, 16, 16, 16]		1.432e-07	1.467e-12
2	[17, 17, 17, 18, 17, 17, 17]		1.745e-08	1.050e-11
3	[17, 17, 17, 17, 17, 17, 17]		2.402e-08	1.213e-11
4	[38, 36, 44, 4, 33, 23, 23]	✔	6.479e-01	5.523e-16
5	[42, 18, 33, 29, 15, 49, 45]	✔	8.494e-10	4.608e-39
6	[51, 51, 10, 37, 38, 42, 16]	✔	9.999e-01	4.513e-38

Neighborhood refinements
—

Best solution [17, 17, 17, 17, 17, 17, 17] found in 32.063s
Optimal solution: [15, 5, 17, 19, 11, 15, 16]

Table 2: Correlation execution data, target 10^{-7} (1)

IT. #	CONFIGURATION	GENERATED	ERROR	PREDICTED
0	[12, 12, 12, 23, 12, 12, 12]		1.021e-05	1.100e-12
1	[14, 14, 14, 14, 14, 14, 14]		1.438e-06	2.948e-11
2	[16, 16, 16, 16, 16, 16, 16]		1.432e-07	6.097e-10
3	[18, 18, 18, 18, 18, 18, 18]		6.520e-09	1.195e-08
4	[20, 6, 48, 5, 22, 36, 4]	✔	4.753e-01	7.641e-03
5	[22, 36, 35, 44, 43, 5, 12]	✔	2.100e-03	7.715e+18
6	[40, 29, 52, 47, 11, 5, 16]	✔	2.095e-03	3.381e-32
7	[14, 13, 44, 24, 36, 27, 46]	✔	2.059e-08	7.448e+00

Neighborhood refinements

—

Best solution [18, 18, 18, 18, 18, 18, 18] found in 35.124s

Optimal solution: [15, 5, 17, 19, 11, 15, 16]

Table 3: Correlation execution data, target 10^{-7} (2)

IT. #	CONFIGURATION	GENERATED	ERROR	PREDICTED
0	[29, 41, 29, 36, 4, 19, 40]	✔	1.595e+09	8.890e-22
1	[46, 28, 14, 29, 32, 18, 33]	✔	5.725e-07	3.666e-28
2	[13, 13, 21, 42, 11, 40, 11]	✔	2.722e-06	1.336e-19
3	[4, 4, 4, 4, 4, 4, 4]		1.000e+00	6.709e-08
4	[31, 48, 36, 26, 6, 44, 44]	✔	8.524e-06	1.875e-66
5	[44, 5, 20, 10, 24, 36, 9]	✔	3.246e-02	2.133e-23

IT. #	CONFIGURATION	GENERATED	ERROR	PREDICTED
6	[12, 12, 12, 29, 12, 12, 12]		1.022e-05	2.269e-09
7	[5, 5, 5, 48, 5, 5, 5]		9.999e-01	8.523e-08
8	[15, 15, 15, 15, 15, 15, 15]		4.230e-07	5.769e-09
9	[14, 14, 14, 28, 14, 14, 14]		6.348e-07	7.635e-08
10	[17, 17, 17, 17, 17, 17, 17]		2.402e-08	4.195e-11
11	[8, 8, 35, 21, 11, 20, 43]	✔	4.851e-05	1.403e-01
12	[43, 49, 44, 11, 12, 47, 12]	✔	6.021e-03	2.975e-11
13	[28, 5, 20, 38, 47, 8, 41]	✔	1.594e+09	4.134e-01
14	[26, 41, 27, 45, 11, 33, 19]	✔	3.984e-08	2.525e-17

Neighborhood refinements

—

Best solution [17, 17, 17, 17, 17, 17, 17] found in 63.927s

Optimal solution: [15, 5, 17, 19, 11, 15, 16]

Table 4: Correlation execution data, target 10^{-7} (3)

IT. #	CONFIGURATION	GENERATED	ERROR	PREDICTED
0	[5, 5, 5, 6, 5, 5, 5]		1.000e+00	6.054e-08
1	[29, 46, 11, 16, 11, 21, 32]	✔	9.999e-01	1.178e-36
2	[13, 13, 13, 13, 13, 13, 13]		8.293e-06	3.030e-09
3	[9, 9, 9, 23, 9, 9, 9]		1.000e+00	2.692e-09
4	[15, 15, 15, 15, 15, 15, 15]		4.230e-07	2.252e-11
5	[17, 17, 17, 23, 17, 17, 17]		1.193e-08	1.703e-12
6	[16, 30, 21, 35, 29, 38, 50]	✔	1.704e-09	2.047e-07
7	[49, 31, 7, 31, 8, 48, 40]	✔	9.999e-01	1.005e-14
8	[6, 39, 33, 33, 6, 43, 24]	✔	8.375e-04	1.912e-01
9	[28, 50, 42, 52, 13, 5, 26]	✔	2.094e-03	3.971e-11

Neighborhood refinements

—

Best solution [17, 17, 17, 23, 17, 17, 17] found in 41.577s
Optimal solution: [15, 5, 17, 19, 11, 15, 16]

Table 5: Correlation execution data, target 10^{-7} (4)

IT. #	CONFIGURATION	GENERATED	ERROR	PREDICTED
0	[39, 5, 12, 38, 26, 34, 24]	✔	1.020e-05	1.953e-01
1	[5, 5, 5, 46, 5, 5, 5]		9.999e-01	8.222e-08
2	[12, 12, 12, 49, 12, 12, 12]		1.022e-05	1.318e-11
3	[7, 33, 9, 20, 44, 22, 14]	✔	9.999e-01	1.127e+03
4	[31, 39, 5, 43, 38, 48, 52]	✔	9.999e-01	2.506e+03
5	[20, 38, 47, 13, 32, 27, 12]	✔	5.944e-06	5.054e+02
6	[16, 16, 16, 32, 16, 16, 16]		3.011e-08	8.216e-08
7	[14, 44, 43, 15, 31, 4, 17]	✔	9.999e-01	1.019e-02
8	[52, 24, 13, 46, 5, 48, 12]	✔	3.408e-05	6.287e+08
9	[21, 11, 5, 25, 19, 18, 42]	✔	9.999e-01	3.379e-06
10	[13, 16, 14, 37, 39, 15, 47]	✔	8.924e-07	8.366e-25

Neighborhood refinements

—

Best solution [16, 16, 16, 32, 16, 16, 16] found in 47.813s
Optimal solution: [15, 5, 17, 19, 11, 15, 16]

Table 6: Correlation execution data, target 10^{-7} (5)

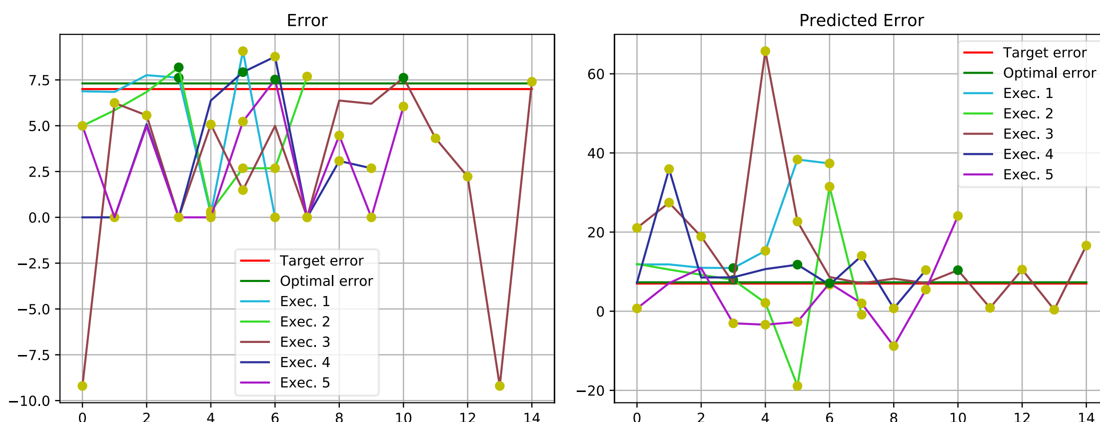


Figure 13: Error and predicted error trends for each execution. Green dots correspond to the best solutions, yellow dots are generated solutions.

The five tables above contain some interesting data points to discuss. First and foremost, it's evident that the script cannot reach the optimal solution without changing the implementation of the Variable Graph or at least without using a neighborhood refinement step or a local search. In fact, this data is a perfect representation of the issue discussed in chapter 5.3 of an overly constraining Variables Graph, since the generic structure of any solution of the correlation benchmark presented there has proven to be applicable to each of the solutions reported here.

Another interesting point is the change in behavior of the regressor between generated and non-generated solutions, as in the first case it tends to predict values that can potentially deviate from the actual error by many orders of magnitude. Generated configurations are generally very heterogeneous in their values and almost always absurd for a human observer, presenting both very precise (high values) and extremely inaccurate (low values) variables and, although valid in theory, they don't take into account the relations among variables and are the result of an incorrect assumption that even a single high precision variable can increase the overall final precision. The regressor doesn't have access to any semantic knowledge about the relationships among variables and the training set is too small to try to build one itself, with the result that it predicts values that are too optimistic and thus wrong. Nevertheless, it's not the machine learning model that is inherently wrong, but only its perception of some configurations. Since the semantic

knowledge is (in part) enforced through constraints by the optimization model, all queries submitted to the regressor during the optimization process result in more precise predictions, although still affected by the scarce training set.

It's important to emphasize that, with an average of 37 seconds per execution, this script manages to find a solution in one-fifth of the time in comparison with state-of-the-art tools such as FpTuning and, despite the impossibility of reaching the actual optimum, it tends to synthesize comparable solutions in terms of bits sum:

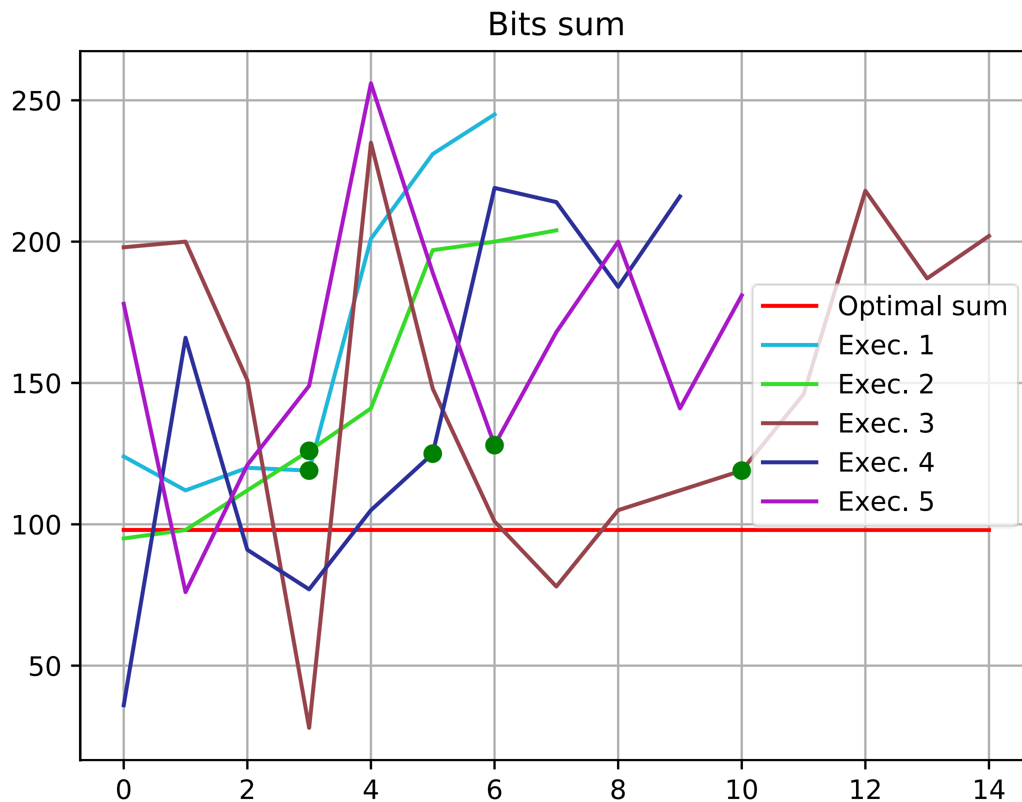


Figure 14: Bits sum trends for each execution. The red line identifies the sum of the optimal solution, while the green dots identify the sums for each best solution

As shown in the figure above, all the best solutions have a bits sum between 119 and 128, with two executions settling at the lowest extremity of the interval. More examples are reported in appendices III and III.

6.2 Neighborhood search Tests

This section covers the differences between the use of a neighborhood search based on interpolation of labels (Radial Base Function on the error) and a neighborhood search based on benchmark executions, in particular highlighting what effects they have on the solution after the optimization cycle. The following tables refer to an execution of the correlation benchmark with a target error of 10^{-5}.

IT. #	CONFIGURATION	GENERATED	ERROR	PREDICTED
0	[10, 10, 10, 17, 10, 10, 10]		1.000e+00	2.031e-10
1	[12, 12, 12, 12, 12, 12, 12]		2.545e-05	8.492e-08
2	[15, 15, 15, 20, 15, 15, 15]		1.369e-07	4.777e-10
3	[50, 39, 33, 29, 8, 6, 29]	✔	1.594e+09	9.435e-01
4	[47, 7, 27, 9, 44, 6, 36]	✔	1.629e+09	2.525e+00
5	[13, 45, 50, 14, 46, 49, 22]	✔	1.058e-06	1.735e-03
6	[9, 24, 48, 52, 40, 6, 42]	✔	1.598e+09	4.398e+09

Neighborhood refinements
—

Best solution [15, 15, 15, 20, 15, 15, 15] found in 23.780s

Neighborhood refinements (version 1)
[15, 15, 15, 17, 15, 15, 15]
[15, 15, 15, 15, 15, 15, 15]

Best solution [15, 15, 15, 15, 15, 15, 15] found in 24.110s

Neighborhood refinements (version 2)
[14, 15, 15, 18, 14, 13, 15]
[14, 12, 15, 18, 14, 11, 14]
[12, 11, 15, 18, 14, 9, 14]
[12, 11, 12, 16, 14, 7, 12]
[12, 11, 11, 14, 12, 6, 11]

Best solution [14, 15, 15, 18, 14, 13, 15] found in 32.897s
Optimal solution: [13, 4, 13, 13, 11, 12, 11]

Table 7: Correlation execution data, target 10^{-5}, all neighborhood search versions

The first version of the neighborhood search (remember it generates neighbors and filters by bits sum and also by adhesion to the Variables Graph constraints) improves the solution found at the end of phase one by lowering one variable to the same value of the others, in line with the solution structure seen back in chapter 5.3. This version doesn't execute the benchmark to check if the solution is feasible, making it extremely fast, but it is also limited in what kind of improvements it can make. In fact, it won't work if the solution at the end of the previous phase doesn't already respect the constraints extracted from the Variables Graph. For instance, there are no chances to find a viable neighbor starting from a configuration such as [13, 16, 12, 19, 14, 13,15], since even if all values were changed, the combined probability of each of them increasing/decreasing in the right quantity is virtually zero.

The second version is a lot more flexible and could potentially find the optimal solution starting from a Variables Graph-compliant configuration. It imposes fewer constraints on the neighbors and generates a lot more possibilities with lower bits sum than the original solution but, unlike for the first, this version cannot be checked by only interpolating the error from the dataset. The reason is that since it is not bound to a limited pool while generating neighbors, it could end up settling on a configuration too far away from the original to correctly interpolate. This is demonstrated by the fact that among all the neighbors reported in Table 7, in the list generated by the second version of the

algorithm, only the first has been proven to actually be feasible, while all the subsequent configurations, although feasible by interpolation, were discarded after executing the benchmark. The necessity of running the benchmark until a feasible neighbor is found (the original solution in the worst scenario), and the associated increase in execution time, the impact of which depends on the maximum number of iterations, is the greatest drawback of this version.

6.3 Active Learning Trend Tests

To test how Active Learning modified the accuracy and stability of the ML model, at each iteration of the optimization cycle the following metrics are calculated:

- $MAE = \frac{1}{n}\sum_{i=1}^{n}|y_i - \gamma_i|$
- $MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \gamma_i)^2$
- $MSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \gamma_i)^2}$

For the regressor, where n is the number of test entries, y_i is the output value and γ_i is the expected value. Instead, for the classifier:

- $Accuracy = \frac{T1+T0}{T1+T0+F1+F0}$
- $Recall = \frac{T1}{T1+F0}$
- $Precision = \frac{T1}{T1+F1}$

With $T1$ being all entries of class 1 actually classified as 1, TO being all entries of class 0 actually classified as 0, $F1$ being all entries of class 0 classified as 1 and F0 being the opposite. The precision can be interpreted as the ability of the classifier not to label as 1 a sample that is actually a 0. The recall is intuitively the ability of the classifier to find all positive examples, while the accuracy specifies the ability of the classifier to correctly label all samples.

Below are listed four examples of executions and their respective metrics graphs.

IT. #	CONFIGURATION	GENERATED	ERROR	PREDICTED
0	[19, 19, 19, 19]		8.497e-11	3.481e-16
1	[21, 21, 21, 21]		5.547e-12	5.762e-16
2	[26, 26, 26, 26]		4.434e-15	2.654e-16
3	[16, 16, 16, 16]		4.147e-09	1.355e-16
4	[12, 12, 12, 12]		1.292e-06	5.451e-16
5	[14, 14, 14, 14]		7.804e-08	9.867e-16
6	[4, 4, 34, 4]		7.754e-03	5.605e-16
7	[5, 5, 27, 5]		2.287e-03	2.798e-16
8	[6, 6, 47, 6]		6.270e-04	8.483e-16
9	[18, 18, 18, 18]		3.009e-10	2.401e-16
10	[23, 23, 23, 23]		3.603e-13	4.086e-16
11	[22, 22, 22, 22]		9.666e-13	4.275e-16
12	[28, 28, 28, 28]		2.822e-16	2.970e-16
13	[7, 7, 51, 10]	✔	1.020e-04	1.201e-09
14	[24, 24, 24, 24]		6.831e-14	1.399e-18
15	[15, 15, 15, 15]		1.950e-08	9.211e-18
16	[17, 17, 17, 17]		1.371e-09	1.320e-17

Neighborhood refinements

—

Best solution [28, 28, 28, 28] found in 52.407s

Optimal solution: [27, 24, 28, 28]

Table 8: Convolution execution data, target 10^{-15}

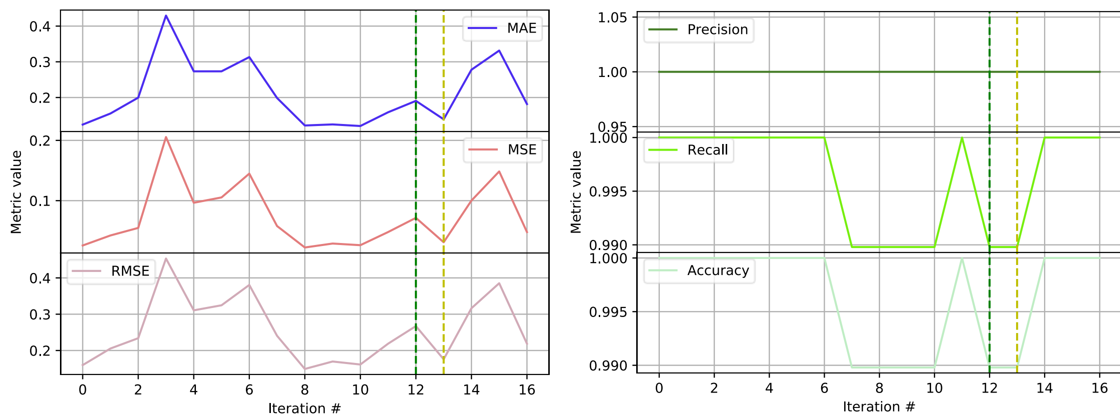


Figure 15: Regressor metrics (left), Classifier metrics (right) (1)

IT. #	CONFIGURATION	GENERATED	ERROR	PREDICTED
0	[21, 21, 24, 21]		1.065e-12	8.929e-21
1	[22, 22, 22, 22]		9.666e-13	1.300e-21
2	[34, 34, 34, 34]		6.441e-20	8.239e-21
3	[19, 19, 19, 19]		8.497e-11	9.525e-22
4	[24, 24, 24, 24]		6.831e-14	1.880e-21
5	[30, 30, 30, 30]		1.920e-17	8.129e-21
6	[31, 31, 31, 31]		4.593e-18	3.384e-21
7	[35, 35, 35, 35]		2.241e-20	4.389e-21
8	[23, 23, 23, 23]		3.603e-13	9.253e-22
9	[28, 28, 28, 28]		2.822e-16	5.968e-21
10	[32, 32, 32, 32]		9.489e-19	1.941e-21
11	[38, 25, 47, 20]	✔	1.161e-12	1.378e+26
12	[46, 46, 46, 46]		4.472e-27	7.735e-21
13	[29, 29, 29, 29]		6.966e-17	3.662e-21
14	[33, 33, 33, 33]		2.841e-19	7.175e-24
15	[36, 36, 38, 36]		1.547e-21	5.851e-25
16	[10, 43, 23, 39]	✔	1.732e-06	1.829e-06

Neighborhood refinements
—

Best solution [36, 36, 38, 36] found in 52.410s
Optimal solution: [36, 34, 36, 36]

Table 9: Convolution execution data, target 10^{-20}

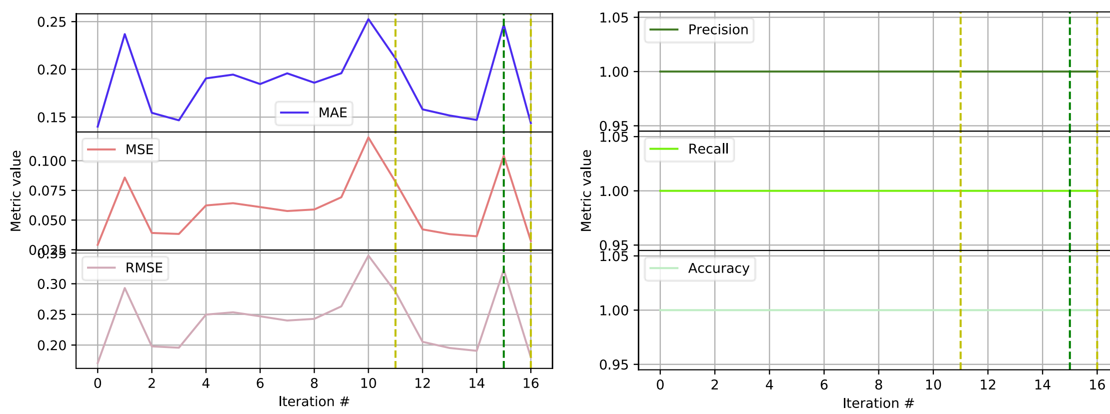


Figure 16: Regressor metrics (left), Classifier metrics (right) (2)

IT. #	CONFIGURATION	GENERATED	ERROR	PREDICTED
0	[5, 5, 5, 5, 5, 5, 5]		1.000e+00	1.133e-06
1	[6, 6, 6, 51, 6, 6, 6]		9.999e-01	8.622e-07
2	[9, 50, 32, 46, 34, 11, 25]	✔	3.973e+04	1.414e-39
3	[48, 32, 33, 32, 31, 43, 50]	✔	1.605e-15	1.138e-38
4	[16, 16, 16, 28, 16, 16, 16]		3.011e-08	9.791e-06
5	[15, 15, 15, 18, 15, 15, 15]		1.562e-07	7.228e-06
6	[13, 13, 13, 18, 13, 13, 13]		2.718e-06	7.723e-06
7	[10, 10, 10, 10, 10, 10, 10]		1.000e+00	9.695e-06
8	[8, 8, 8, 8, 8, 8, 8]		4.161e+04	3.758e-07

Neighborhood refinements

—

Best solution [13, 13, 13, 18, 13, 13, 13] found in 39.162s
Optimal solution: [13, 4, 13, 13, 11, 12, 11]

Table 10: Correlation execution data, target 10^{-5}

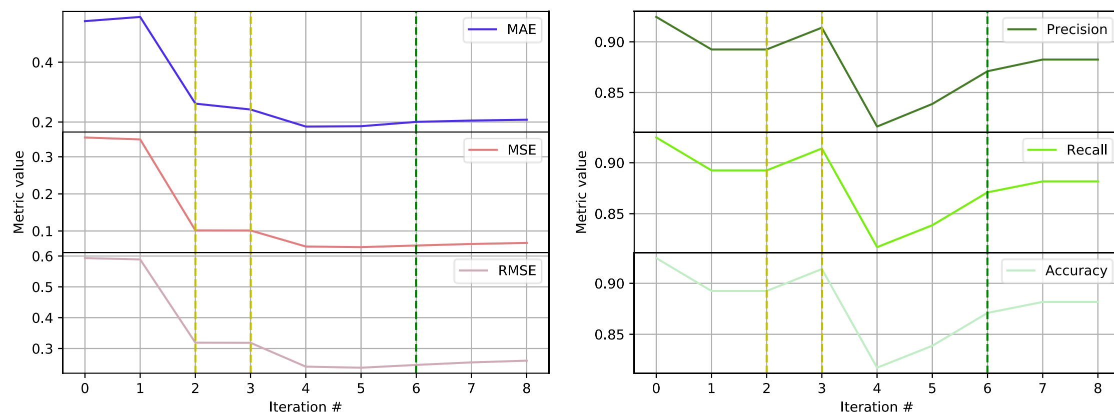


Figure 17: Regressor metrics (left), Classifier metrics (right) (3)

IT. #	CONFIGURATION	GENERATED	ERROR	PREDICTED
0	[12, 12, 12, 12, 12, 12, 12]		2.545e-05	6.166e-14
1	[15, 15, 15, 15, 15, 15, 15]		4.230e-07	2.677e-13
2	[17, 17, 17, 17, 17, 17, 17]		2.402e-08	3.152e-15
3	[19, 19, 19, 19, 19, 19, 19]		2.268e-09	1.822e-15
4	[20, 20, 20, 20, 20, 20, 20]		3.980e-10	2.834e-14
5	[23, 23, 23, 43, 23, 23, 23]		2.070e-12	1.498e-15
6	[22, 22, 22, 45, 22, 22, 22]		1.284e-11	1.193e-15
7	[21, 21, 21, 38, 21, 21, 21]		2.865e-11	1.295e-15
8	[14, 14, 14, 24, 14, 14, 14]		6.346e-07	5.064e-11
9	[18, 18, 18, 32, 18, 18, 18]		2.427e-09	3.601e-11

Neighborhood refinements

—
Best solution [21, 21, 21, 38, 21, 21, 21] found in 42.392s
Optimal solution: [20, 13, 22, 23, 19, 20, 20]

Table 11: Correlation execution data, target 10^{-10}

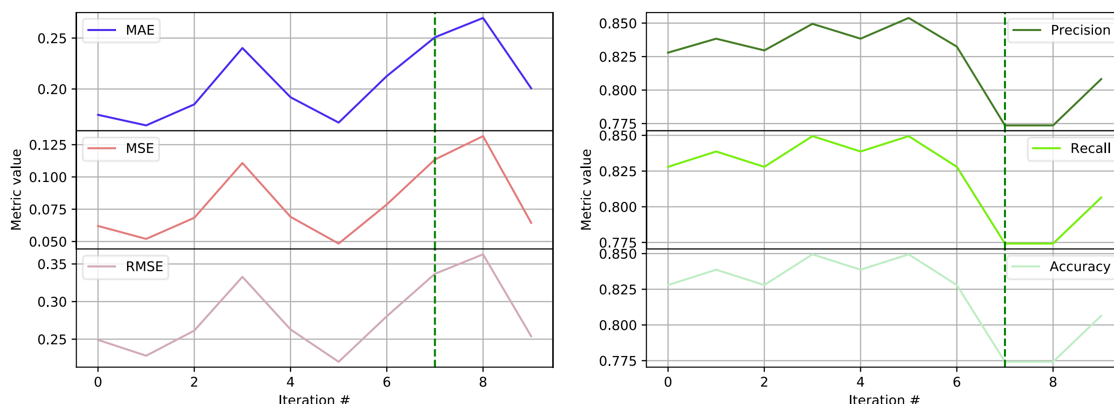


Figure 18: Regressor metrics (left), Classifier metrics (right) (4)

Let's start by clarifying that it's difficult to induce a general trend of any of these metrics within so few iterations and, consequently, to give a clear evaluation of the training rate of both the regressor and the classifier. What is most interesting is that MAE, MSE, and RMSE seem to stay confined to small numbers, often reaching really low values that would suggest stability and solidity of the regressor model, but their seemingly erratic behavior prevents from telling whether they tend to converge to low values or not. Moreover, the data presented in the tables above don't reflect the values the metrics portray, with differences of many orders of magnitudes between predicted errors and actual values, even without considering the generated solutions. Since the test set used to extrapolate all metrics is extracted from the initial dataset, it can be only assumed that such a deviation of the experimental data from the regressor metrics is symptomatic of a case of overfitting. It is in part compensated by the optimization model and the execution of the benchmark, but there are rare cases where the initial training makes the regressor too unstable to be compensated for (see Appendix IV). Since the initial training set must be kept as small as possible, the methodology to avoid the overfitting can be unfolded on two different fronts:

- Creating a more incisive initial training set, focusing only on the configurations that are meaningful or critical, such as those with a very low or very high bits count.
- Use some domain-bound heuristic to generate the neighbors used for the retraining, in order to only select those configurations that may have a higher impact on the performances of the regressor.

The classifier suffers a lot less than the regressor from the gaunt dataset and tends to better classify errors than the regressor to correctly predict them, although this might be partially due to the fact that the two possible classes cover a much broader domain than the single values. There have been some experiments where the classifier apparently worked perfectly, with a precision and accuracy of 100% (see Figure 16) that given the very low number of samples used to train the decision tree are very probably the result of overfitting. However, for the same reason above, it was impossible to effectively prove whether or not it was, in fact, overfitting.

7 Conclusions

The main focus of this paper has been to find a way to enhance the Active Learning process and to improve the stability and precision of both the optimization model and the machine learning models. To achieve the final result the development has followed the steps below:

1. Refactor of the pre-existing architecture to only keep the fundamentals modules and functionalities, removing what seemed to be superfluous.
2. Implementation of a new sampling method for the initial training and test sets.
3. Implementation of a more compact and efficient way to build the optimization model.
4. Implementation and test of various new methods for generating additional samples for the retraining.
5. Implementation of two versions of a neighborhood search and comparison between the two.
6. Multiple test runs and tweaking of the hard-coded parameters.

The first issue was encountered when sampling the initial training and test set. Although the script tries to sample examples that are for the most part informative, the initial dataset is too random and shallow to be effective. It has been generated by applying the Latin hyper-cube sampling method, which should technically guarantee that there are no duplicates, but somehow on 1000 sampled entries at every execution, at least 30 of them are removed as duplicates. Moreover, although this method covers more or less uniformly the configurations space, most of them are "semantically dead". To explain this concept, let's consider a generic configuration $[V0, \dots, Vk, \dots, Vn]$: any semantically dead configuration is one where there is at least one couple of variables Vx and Vy, $\text{len}(Vx) \gg \text{len}(Vy)$ or $\text{len}(Vx) \ll \text{len}(Vy)$, e.g. for convolution $[12, 11, 9, 48]$. Such a configuration has no semantic meaning since it specifies a program where at least one variable is extremely more precise than another, which is an impossible case. If the two

variables are related, for instance through an assignment, the error is even bigger. These configurations can be easily spotted, as they usually are labeled with an error that can reach orders of magnitude higher than 10^6. Of course, removing these configurations from the training set would make the regressor extremely imprecise when it comes to predicting their errors, but the Variables Graph already forces the optimization model to only consider those configurations that are more significant and thus removing the need for the regressor to be able to precisely predict the error for those that are semantically dead.

The other big issue to address is overfitting. Both the classifier and the regressors seem to suffer greatly from this problem and, given the necessity to keep the training set as small as possible, there is no easy solution. The first aid comes from building a better initial dataset and, although it doesn't solve the issue per se, it reduces how critical the problem is, at least initially. Technically speaking, removing all semantically dead configurations from the initial dataset increases overfitting and it renders the regressor unable to predict effectively all possible configurations of the input space. Nevertheless, since the actual domain of interest is only composed of those most significant solutions, the entity of the issue is significantly decreased. The fact remains that the core solution to overfitting should be located in the Active Learning cycle.

Given a solution and its relative error, the generation of a set of neighbors and its labeling by means of interpolating algorithms has proven to be really helpful in increasing the effectiveness of the retraining stages, but while it is true that it's not necessary for the regressor to be extremely precise, there is the possibility that without limiting how approximated examples influence its behavior, the regressor's performances could get worse over time. It is important for future developments to focus on both the generation of neighbors and the generation of alternative solutions, trying to make them all be the most informative possible, in accordance with the philosophy of Active Learning.

Lastly, the other big issue that should be addressed is the Variables Graph or, to be precise, how the relationships between variables are extracted from it. In the current state, these relationships are too strict and force the optimization model to only consider a limited number of solutions that are worse than the optimal solution found with other automatic methods. There has been no possibility to check whether the given Variables

Graphs are right or not, but it could be possible that some variables have been related to each other when it wasn't necessary. For instance, consider the following list of optimal solutions in the case of the correlation benchmark:

TARGET	OPTIMAL SOLUTION
10^N	**[V0, V1, V2, V3, T4, T5, T6]**
1	[7, 4, 12, 7, 6, 10, 4]
2	[7, 4, 12, 11, 6, 10, 5]
3	[7, 4, 12, 12, 6, 10, 7]
5	[13, 4, 13, 13, 11, 12, 11]
7	[15, 5, 17, 19, 11, 15, 16]
10	[20, 13, 22, 23, 19, 20, 20]
12	[24, 24, 24, 25, 24, 24, 24]
15	[29, 25, 31, 31, 27, 29, 29]
20	[37, 26, 39, 39, 36, 36, 38]
25	[48, 37, 48, 48, 46, 47, 47]

Table 12: All optimal solutions for the correlation benchmark

The first thing that becomes clear is that the relationship $\text{len}(V0) \leq \text{len}(V1)$ is most probably wrong, as well as the relationship $\text{len}(V2) = \text{len}(T5)$. Since the Variables Graph proved to be a really powerful aid to both the Active Learning cycle and the optimization phase, it is mandatory to explore more deeply its potential and try to correct those parts that are currently limiting the quality of the solutions.

In conclusion, the current state of this procedure using a hybrid approach between Mathematical Programming and Active Learning is not fully developed yet, but by focusing further efforts on making Active Learning a viable tool to use in combination with an optimization model, it could be possible to develop a system capable of substituting the current state-of-the-art technologies, while also being equally reliable.

References

Hauser, J. (2019, November 6). *Berkeley SoftFloat*. Retrieved from jhauser: http://www.jhauser.us/arithmetic/SoftFloat.html

IBM. (2019, 11 13). *IBM Decision Optimization CPLEX Modeling for Python -- IBM Decision Optimization CPLEX Modeling for Python (DOcplex) V2.11 documentation*. Retrieved from ibmdecisionoptimization: https://ibmdecisionoptimization.github.io/docplex-doc/index.html

IBM. (2019, 11 13). *Overview of mathematical programming -- IBM Decision Optimization CPLEX Modeling for Python (DOcplex) V2.11 documentation*. Retrieved from ibmdecisionoptimization: https://ibmdecisionoptimization.github.io/docplex-doc/mp.html

Konyushkova, K., & Sznitman, R. (2017). *Learning Active Learning from Data.*

Kyu Hyun, C. (2017). *Explaining Active Learning Queries.*

Mach, S., Rossi, D., Tagliavini, G., Marongiu, A., & Benini, L. (2018). *A Transprecision Floating-Point Architecture for Energy-Efficient Embedded Computing.*

Malossi, A. C., Schaffner, M., Molnos, A., Gammaitoni, L., Tagliavini, G., Emerson, A., . . . Wehn, N. (2018). *The Transprecision Computing Paradigm: Concept, Design, and Applications.*

Martello, S. (2014). Mathematical Programming. In S. Martello, *Operative Research* (p. 17). Bologna: Esculapio.

Milano, M., & Lombardi, M. (2019, 11 14). *Empirical Model Learning | Embedding Machine Learning Models in Optimization*. Retrieved from emlopt: https://emlopt.github.io/

Milano, M., Lombardi, M., & Bartolini, A. (2017). Empirical Decision Model Learning. *Artificial Intelligence*, 343-367.

OPRECOMP. (2019, November 5). *PARTNERS - OPRECOMP*. Retrieved from oprecomp: http://oprecomp.eu/consortium/

OPRECOMP. (2019, November 5). *PROJECT DESCRIPTION - OPRECOMP*. Retrieved from oprecomp: http://oprecomp.eu/project-description/

Settles, B. (2009). *Active Learning Literature Survey.* University of Wisconsin-Madison.

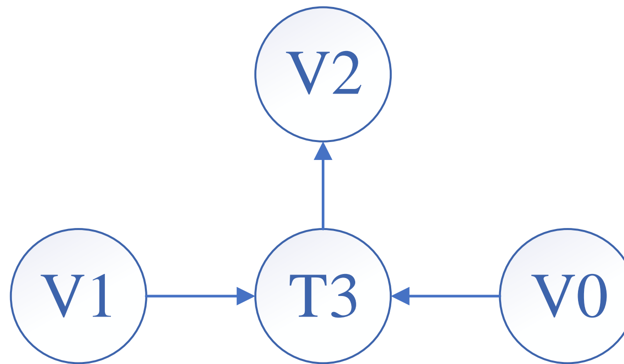
Tagliavini, G., Marongiu, A., & Benini, L. (2018). *FlexFloat: A Software Library for Transprecision Computing*. IEEE.

The GNU MPFR Library. (2019, November 6). Retrieved from mpfr: https://www.mpfr.org/

Xin, H., Zhao, K., & Chu, X. (2019). *AutoML: A Survey of the State-of-the-Art.* Cornell University.

Appendices

I. Convolution Variables Graph



From which:

- $\mathrm{len}(V0) \leq \mathrm{len}(T3)$;
- $\mathrm{len}(V1) \leq \mathrm{len}(T3)$;
- $\mathrm{len}(T3) \leq \mathrm{len}(V2)$;
- $\mathrm{len}(T3) = \min(\mathrm{len}(V0), \mathrm{len}(V1))$.

From which it can be derived that:

- $\mathrm{len}(V0) = \mathrm{len}(T3)$;
- $\mathrm{len}(V1) = \mathrm{len}(T3)$;
- $\mathrm{len}(T3) \leq \mathrm{len}(V2)$

Given the generic structure $[V0, V1, V2, T3]$, it can be deduced the structure $[a, a, b, a]$ with $a \leq b$.

II. Convolution executions

Target error: 10^{-5}

Execution 1.

IT. #	CONFIGURATION	GENERATED	ERROR	PREDICTED
0	[5, 5, 5, 5]		1.794e-02	7.446e-06
1	[6, 6, 6, 6]		5.012e-03	8.640e-08
2	[9, 9, 9, 9]		7.216e-05	5.711e-06
3	[7, 7, 7, 7]		1.193e-03	1.043e-07
4	[8, 8, 8, 8]		2.644e-04	1.096e-06
5	[10, 10, 10, 10]		2.292e-05	3.748e-08
6	[11, 11, 11, 11]		5.456e-06	9.633e-09
7	[23, 37, 10, 19]	✔	1.902e-05	1.145e-23
8	[4, 32, 28, 37]	✔	6.448e-03	8.272e-01
9	[17, 48, 21, 33]	✔	1.254e-10	6.131e-09
10	[41, 44, 40, 45]	✔	2.108e-23	3.857e-45

Neighborhood refinements
—

Best solution [11, 11, 11, 11] found in 39.238s
Optimal solution: [10, 8, 12, 10]

Execution 2.

IT. #	CONFIGURATION	GENERATED	ERROR	PREDICTED
0	[5, 5, 5, 5]		1.794e-02	2.980e-06
1	[6, 6, 6, 6]		5.012e-03	5.526e-08
2	[7, 7, 7, 7]		1.193e-03	1.393e-06
3	[8, 8, 8, 8]		2.644e-04	4.870e-07
4	[4, 4, 23, 4]		7.754e-03	5.501e-06
5	[9, 9, 9, 9]		7.216e-05	9.901e-07
6	[10, 10, 10, 10]		2.292e-05	1.214e-06
7	[11, 11, 11, 11]		5.456e-06	2.140e-09
8	[42, 22, 32, 38]	✔	2.459e-15	6.887e-26
9	[47, 10, 4, 17]	✔	8.170e-02	7.772e-17
10	[48, 52, 18, 35]	✔	3.009e-10	3.739e-03
11	[35, 13, 37, 34]	✔	1.855e-09	8.999e-11

Neighborhood refinements

—

Best solution [11, 11, 11, 11] found in 38.168s

Optimal solution: [10, 8, 12, 10]

Execution 3.

IT. #	CONFIGURATION	GENERATED	ERROR	PREDICTED
0	[6, 6, 6, 6]		5.012e-03	3.156e-06
1	[7, 7, 7, 7]		1.193e-03	3.597e-07
2	[12, 12, 12, 12]		1.292e-06	6.077e-06
3	[8, 8, 8, 8]		2.644e-04	6.097e-07
4	[46, 31, 10, 14]	✔	1.902e-05	1.104e-04

5	[9, 9, 9, 9]		7.216e-05	2.315e-08
6	[28, 51, 10, 18]	✔	1.902e-05	1.556e-15

Neighborhood refinements

—

Best solution [12, 12, 12, 12] found in 24.603s

Optimal solution: [10, 8, 12, 10]

Execution 4.

IT. #	CONFIGURATION	GENERATED	ERROR	PREDICTED
0	[7, 7, 7, 7]		1.193e-03	1.995e-06
1	[6, 6, 6, 6]		5.012e-03	4.124e-06
2	[8, 8, 8, 8]		2.644e-04	2.874e-06
3	[9, 9, 9, 9]		7.216e-05	4.965e-07
4	[10, 10, 10, 10]		2.292e-05	1.113e-07
5	[24, 24, 24, 24]		6.831e-14	9.827e-06
6	[11, 11, 11, 11]		5.456e-06	5.055e-07
7	[5, 11, 50, 23]	✔	1.851e-03	1.929e+43
8	[33, 50, 7, 20]	✔	1.076e-03	4.021e-17
9	[37, 35, 24, 30]	✔	6.632e-14	7.334e-01

Neighborhood refinements

—

Best solution [11, 11, 11, 11] found in 32.830s

Optimal solution: [10, 8, 12, 10]

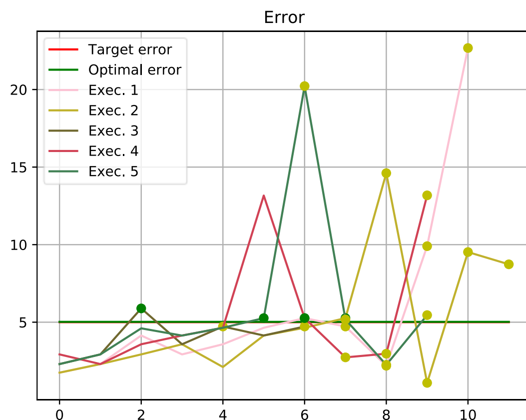
Execution 5.

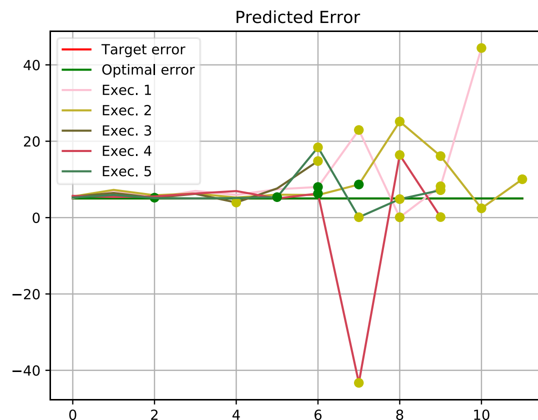
IT. #	CONFIGURATION	GENERATED	ERROR	PREDICTED
0	[6, 6, 6, 6]		5.012e-03	5.789e-06
1	[7, 7, 7, 7]		1.193e-03	1.150e-06
2	[9, 9, 11, 9]		2.490e-05	7.419e-06
3	[8, 8, 10, 8]		7.285e-05	9.716e-06
4	[10, 10, 10, 10]		2.292e-05	7.541e-06
5	[11, 11, 11, 11]		5.456e-06	4.391e-06
6	[34, 43, 45, 48]	✔	6.015e-21	3.975e-19
7	[9, 34, 29, 40]	✔	6.670e-06	7.999e-01
8	[10, 46, 6, 22]	✔	5.626e-03	1.440e-05
9	[50, 50, 36, 9]	✔	3.497e-06	6.547e-08

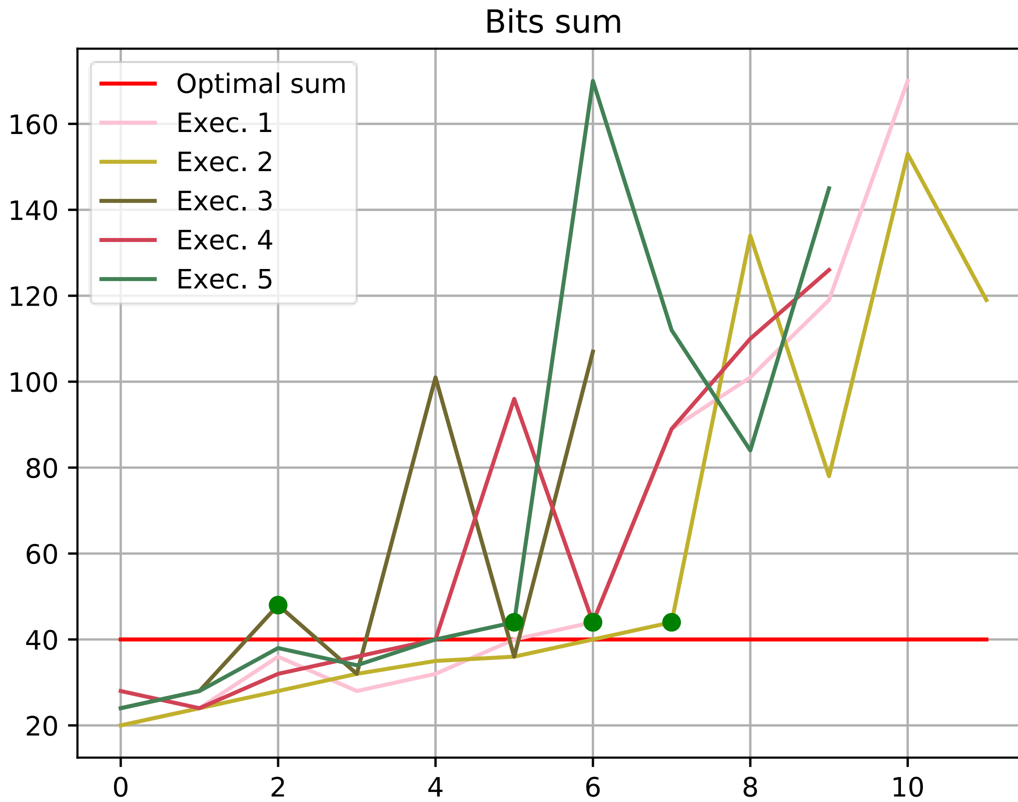
Neighborhood refinements

Best solution [11, 11, 11, 11] found in 33.132s

Optimal solution: [10, 8, 12, 10]







III. Correlation executions

Target error: 10^{-5}

Execution 1.

IT. #	CONFIGURATION	GENERATED	ERROR	PREDICTED
0	[5, 5, 5, 5, 5, 5, 5]		1.000e+00	1.572e-07
1	[15, 38, 45, 48, 36, 13, 38]	✔	5.808e-08	7.680e-42
2	[6, 6, 6, 49, 6, 6, 6]		9.999e-01	4.734e-10
3	[33, 27, 48, 36, 39, 12, 27]	✔	1.338e-07	2.688e-40
4	[6, 27, 14, 11, 17, 44, 39]	✔	7.182e-03	6.869e-22
5	[22, 13, 15, 11, 46, 31, 43]	✔	5.853e-03	3.891e-37

6	[10, 10, 10, 32, 10, 10, 10]		1.000e+00	8.972e-07
7	[13, 23, 11, 9, 28, 21, 26]	✔	1.000e+00	3.625e-29
8	[51, 12, 43, 21, 4, 6, 28]	✔	1.594e+09	5.593e+08
9	[17, 17, 17, 46, 17, 17, 17]		1.200e-08	4.725e-07
10	[13, 13, 13, 13, 13, 13, 13]		8.293e-06	7.076e-09
11	[11, 16, 17, 38, 34, 41, 7]	✔	5.853e-03	2.435e-21
12	[23, 45, 39, 41, 38, 17, 49]	✔	1.277e-10	1.090e+01
13	[7, 44, 27, 24, 30, 4, 26]	✔	9.999e-01	6.023e-11

Neighborhood refinements

—

Best solution [13, 13, 13, 13, 13, 13, 13] found in 62.864s

Optimal solution: [13, 4, 13, 13, 11, 12, 11]

Execution 2.

IT. #	CONFIGURATION	GENERATED	ERROR	PREDICTED
0	[5, 5, 5, 15, 5, 5, 5]		9.999e-01	1.319e-06
1	[12, 12, 12, 37, 12, 12, 12]		1.022e-05	1.165e-10
2	[14, 14, 14, 25, 14, 14, 14]		6.346e-07	1.224e-10
3	[16, 24, 11, 40, 44, 33, 4]	✔	9.999e-01	5.711e-01
4	[43, 34, 18, 41, 19, 34, 33]	✔	2.520e-09	5.264e-08
5	[19, 47, 28, 52, 44, 28, 25]	✔	2.366e-11	1.741e-41
6	[8, 25, 32, 24, 22, 49, 38]	✔	4.852e-05	6.508e-50

Neighborhood refinements

—

Best solution [14, 14, 14, 25, 14, 14, 14] found in 32.025s
Optimal solution: [13, 4, 13, 13, 11, 12, 11]

Execution 3.

IT. #	CONFIGURATION	GENERATED	ERROR	PREDICTED
0	[5, 5, 5, 5, 5, 5, 5]		1.000e+00	6.170e-07
1	[46, 48, 7, 19, 50, 24, 45]	✔	9.999e-01	1.285e-44
2	[6, 6, 6, 45, 6, 6, 6]		9.999e-01	1.858e-07
3	[20, 50, 18, 24, 32, 32, 47]	✔	2.041e-09	1.588e-31
4	[10, 10, 10, 38, 10, 10, 10]		1.000e+00	1.342e-10
5	[47, 51, 31, 36, 13, 19, 43]	✔	9.917e-09	6.316e-01
6	[16, 16, 16, 46, 16, 16, 16]		3.011e-08	2.606e-10
7	[29, 32, 42, 8, 31, 14, 7]	✔	7.803e-03	2.007e+05
8	[4, 4, 4, 50, 4, 4, 4]		9.999e-01	9.249e-06
9	[23, 7, 23, 36, 52, 12, 45]	✔	1.382e-07	4.221e-12
10	[34, 51, 28, 46, 39, 23, 12]	✔	6.726e-07	2.185e-03

Neighborhood refinements
—

Best solution [16, 16, 16, 46, 16, 16, 16] found in 47.563s
Optimal solution: [13, 4, 13, 13, 11, 12, 11]

Execution 4.

IT. #	CONFIGURATION	GENERATED	ERROR	PREDICTED
0	[12, 12, 12, 24, 12, 12, 12]		1.022e-05	5.226e-09
1	[14, 14, 14, 14, 14, 14, 14]		1.438e-06	1.076e-08
2	[9, 25, 16, 25, 45, 47, 46]	✔	3.972e+04	2.683e-06
3	[45, 51, 15, 17, 46, 11, 4]	✔	4.100e+04	1.199e-06
4	[18, 41, 17, 37, 11, 43, 28]	✔	5.606e-08	1.227e+15
5	[5, 5, 5, 5, 5, 5, 5]		1.000e+00	9.799e-06

Neighborhood refinements

—

Best solution [14, 14, 14, 14, 14, 14, 14] found in 28.575s

Optimal solution: [13, 4, 13, 13, 11, 12, 11]

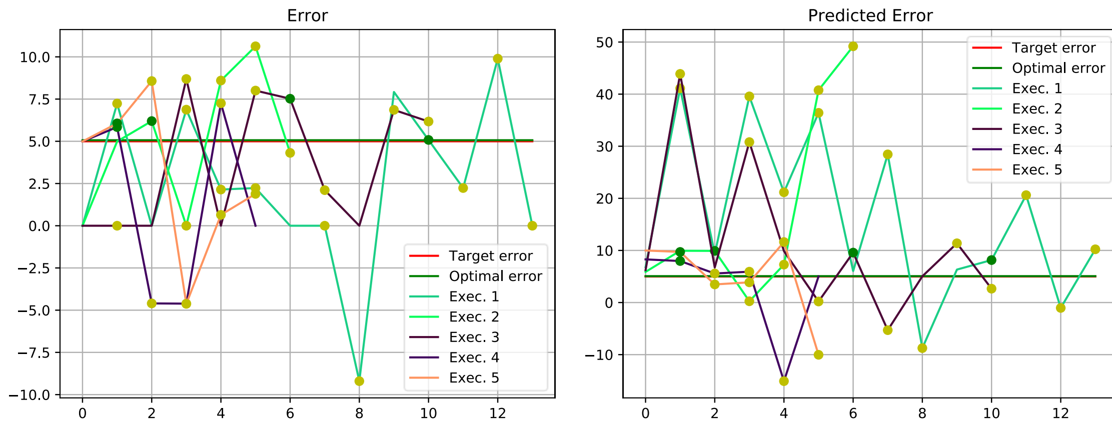
Execution 5.

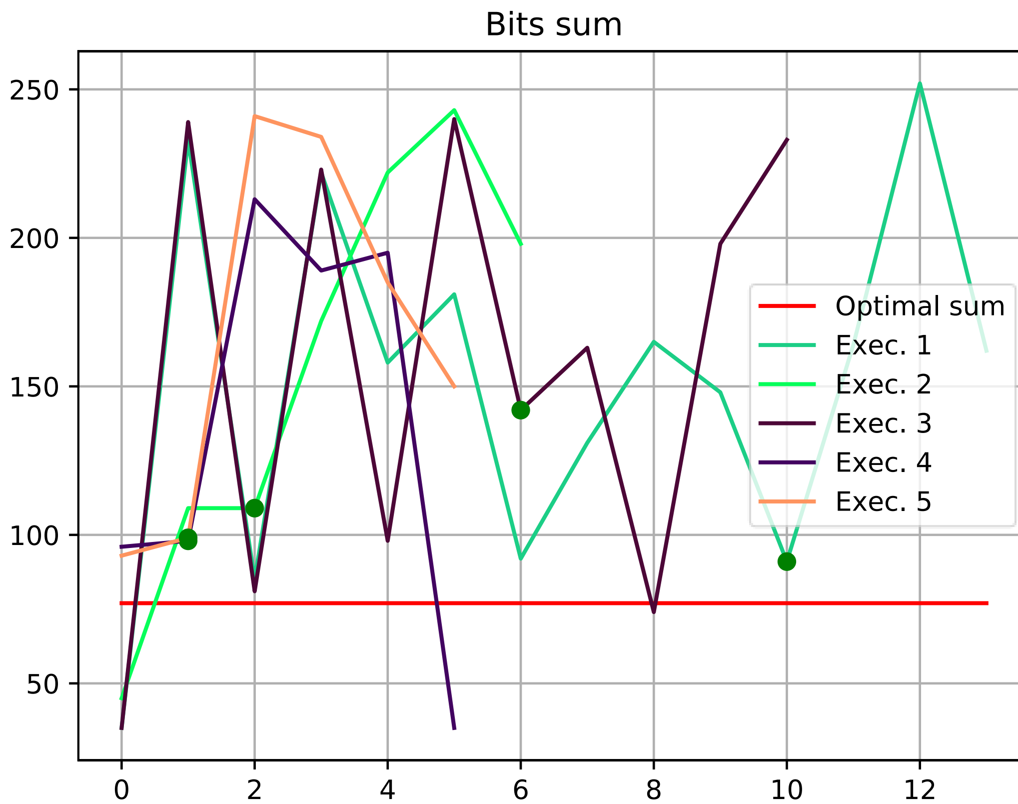
IT. #	CONFIGURATION	GENERATED	ERROR	PREDICTED
0	[12, 12, 12, 21, 12, 12, 12]		1.021e-05	1.031e-10
1	[14, 14, 14, 15, 14, 14, 14]		8.696e-07	1.936e-10
2	[39, 25, 44, 29, 14, 48, 42]	✔	2.706e-09	3.346e-04
3	[5, 35, 28, 34, 50, 44, 38]	✔	4.226e+04	1.331e-04
4	[23, 37, 31, 9, 11, 31, 43]	✔	2.271e-01	2.416e-12
5	[46, 8, 52, 21, 14, 5, 4]	✔	1.302e-02	1.079e+10

Neighborhood refinements

—

Best solution [14, 14, 14, 15, 14, 14, 14] found in 28.919s
Optimal solution: [13, 4, 13, 13, 11, 12, 11]





IV. Convolution – failed initial training

The execution results reported here refer to the convolution benchmark with target error of 10^{-5}.

IT. #	CONFIGURATION	GENERATED	ERROR	PREDICTED
0	[34, 16, 26, 44]	✔	3.481e-11	8.274e-23
1	[7, 45, 33, 42]	✔	1.128e-04	2.250e-40
2	[23, 45, 29, 20]	✔	1.293e-12	8.385e-17
3	[40, 27, 5, 14]	✔	1.918e-02	1.017e-19
4	[13, 4, 52, 40]	✔	5.445e-04	6.824e-20
5	[22, 19, 33, 43]	✔	1.684e-12	1.104e-21
6	[12, 21, 34, 27]	✔	1.221e-07	6.713e-34
7	[7, 37, 27, 51]	✔	1.128e-04	7.447e-33
8	[11, 50, 10, 38]	✔	1.902e-05	4.398e-21
9	[45, 11, 22, 29]	✔	2.556e-08	2.123e-09
10	[50, 28, 52, 35]	✔	7.275e-19	2.206e-37
11	[48, 40, 47, 38]	✔	1.300e-23	4.083e-26
12	[24, 32, 9, 36]	✔	9.652e-05	1.391e-23
13	[16, 10, 32, 5]	✔	7.110e-04	4.686e-07
14	[49, 17, 23, 47]	✔	2.733e-11	1.355e-06
15	[29, 6, 35, 50]	✔	2.203e-05	8.038e-16
16	[32, 46, 40, 18]	✔	9.534e-12	1.414e-10
17	[24, 23, 16, 50]	✔	4.147e-09	3.088e-34
18	[32, 26, 13, 28]	✔	2.898e-07	4.832e-26
19	[36, 22, 6, 15]	✔	5.626e-03	2.121e-20
20	[4, 31, 27, 34]	✔	6.448e-03	1.305e-32

21	[16, 34, 8, 13]	✔	2.901e-04	8.443e-11
22	[52, 38, 35, 24]	✔	2.928e-15	5.065e-43
23	[38, 25, 51, 49]	✔	1.449e-16	1.321e-15
24	[40, 42, 50, 18]	✔	9.534e-12	1.565e-29
25	[25, 18, 24, 50]	✔	6.840e-13	7.563e-29
26	[46, 17, 7, 47]	✔	1.076e-03	9.663e-29
27	[16, 45, 22, 40]	✔	4.133e-10	1.226e-10
28	[9, 9, 50, 36]	✔	9.091e-06	6.447e-12
29	[49, 52, 25, 7]	✔	7.361e-05	5.779e-07
30	[46, 44, 5, 9]	✔	2.071e-02	2.001e-06
31	[9, 29, 19, 40]	✔	6.670e-06	5.393e-07
32	[4, 19, 28, 52]	✔	6.448e-03	1.661e-12
33	[48, 11, 6, 14]	✔	5.626e-03	2.482e-08
34	[22, 24, 36, 48]	✔	1.071e-13	7.966e-19
35	[52, 7, 24, 21]	✔	3.100e-06	3.604e-20
36	[26, 7, 12, 32]	✔	3.708e-06	1.674e-06
37	[6, 51, 24, 27]	✔	4.606e-04	2.379e-20
38	[33, 6, 7, 14]	✔	1.730e-03	1.531e-01
39	[30, 50, 31, 37]	✔	7.734e-18	3.599e-10
40	[45, 13, 9, 48]	✔	9.652e-05	7.935e-10
41	[40, 18, 36, 8]	✔	1.070e-05	9.952e-05
42	[20, 4, 16, 30]	✔	5.429e-04	3.003e-02
43	[50, 22, 51, 16]	✔	2.500e-10	2.945e-09
44	[26, 23, 37, 7]	✔	7.361e-05	2.924e-23
45	[29, 44, 44, 7]	✔	7.361e-05	3.925e-08
46	[32, 15, 30, 13]	✔	2.294e-08	1.226e-01

47	[40, 18, 14, 6]	✔	2.609e-04	4.761e-07
48	[12, 21, 25, 32]	✔	1.221e-07	1.052e-07
49	[43, 37, 50, 20]	✔	1.161e-12	1.987e+02
50	[37, 11, 39, 47]	✔	2.561e-08	2.740e-20
51	[8, 23, 25, 17]	✔	2.736e-05	3.287e-12
52	[10, 42, 24, 5]	✔	6.377e-04	3.131e-21
53	[16, 33, 41, 4]	✔	1.519e-03	3.156e-24
54	[46, 24, 19, 44]	✔	6.352e-11	5.779e-13
55	[9, 13, 13, 30]	✔	7.826e-06	3.413e-11
56	[13, 4, 7, 22]	✔	2.696e-03	5.600e-09
57	[44, 23, 12, 8]	✔	1.070e-05	1.929e-03
58	[50, 8, 23, 29]	✔	7.675e-07	2.321e-20
59	[27, 24, 42, 24]	✔	3.070e-15	1.262e-04
60	[37, 28, 28, 16]	✔	2.500e-10	9.043e-08
61	[7, 34, 19, 6]	✔	3.535e-04	5.686e-08
62	[13, 46, 5, 19]	✔	1.918e-02	1.042e-16
63	[42, 5, 9, 29]	✔	1.332e-04	6.442e-04
64	[47, 36, 4, 6]	✔	8.170e-02	4.301e-02
65	[23, 45, 13, 28]	✔	2.898e-07	4.166e-14
66	[41, 43, 47, 13]	✔	1.178e-08	4.234e-08
67	[20, 18, 14, 42]	✔	7.004e-08	4.312e-19
68	[24, 36, 45, 49]	✔	6.947e-15	2.364e-34
69	[19, 43, 11, 20]	✔	4.309e-06	9.837e-16
70	[23, 25, 45, 19]	✔	5.287e-12	1.570e-09
71	[44, 9, 22, 45]	✔	8.424e-07	3.430e-05
72	[45, 10, 45, 42]	✔	4.064e-07	5.998e-02

73	[51, 27, 22, 10]	✔	1.545e-06	9.763e+03
74	[15, 25, 45, 42]	✔	1.807e-09	1.091e-07
75	[22, 20, 33, 7]	✔	7.361e-05	7.303e-04
76	[4, 28, 34, 5]	✔	6.987e-03	1.066e-05
77	[31, 18, 13, 11]	✔	6.749e-07	1.801e-06
78	[29, 19, 52, 16]	✔	2.500e-10	8.510e-40
79	[19, 13, 7, 16]	✔	1.076e-03	8.949e-09
80	[38, 42, 41, 48]	✔	3.640e-23	1.922e-43
81	[8, 13, 23, 31]	✔	2.736e-05	2.115e-21
82	[29, 50, 44, 47]	✔	6.318e-18	2.424e-16
83	[48, 33, 31, 10]	✔	1.545e-06	1.336e-04
84	[9, 13, 27, 21]	✔	6.670e-06	4.322e-15
85	[26, 5, 51, 44]	✔	2.876e-05	5.241e-29
86	[5, 39, 9, 12]	✔	1.964e-03	2.582e-14
87	[51, 44, 50, 21]	✔	3.359e-13	2.337e-29
88	[31, 8, 20, 31]	✔	7.656e-07	6.859e-41
89	[31, 16, 23, 48]	✔	3.706e-11	1.970e-43
90	[39, 31, 5, 24]	✔	1.918e-02	1.676e-27
91	[51, 20, 47, 5]	✔	7.522e-04	2.415e+12
92	[50, 20, 49, 9]	✔	3.497e-06	3.938e+14
93	[4, 40, 6, 18]	✔	1.661e-02	6.632e-08
94	[36, 41, 13, 39]	✔	2.898e-07	1.212e-03
95	[8, 42, 10, 37]	✔	6.921e-05	7.826e-12
96	[10, 12, 38, 41]	✔	2.037e-06	8.834e-20
97	[27, 49, 47, 39]	✔	1.031e-16	4.730e-24
98	[51, 47, 13, 28]	✔	2.898e-07	1.639e-05

99	[41, 50, 39, 44]		6.132e-23	3.833e-43

Neighborhood refinements

—

No solution found after 287.455s

Optimal solution: [10, 8, 12, 10]