UNIVERSITY POLITEHNICA OF BUCHAREST
FACULTY OF AUTOMATIC CONTROL AND COMPUTERS
COMPUTER SCIENCE AND ENGINEERING DEPARTMENT

Dissertation project

Complexity-Based Code Insights

An approach to understanding how generic computer code can be automatically analyzed using r-Complexity-based code embeddings, for the problem of classifying solutions of competitive programming challenges.

Rares Folea

Scientific advisors:

Radu Iacob, Traian Rebedea

BUCHAREST
2022

UNIVERSITATEA POLITEHNICA DIN BUCUREȘTI
FACULTATEA DE AUTOMATICĂ ȘI CALCULATOARE
DEPARTAMENTUL DE CALCULATOARE

Proiect de disertatie

Analiza codului bazată pe calculul complexității

O abordare pentru înțelegerea modului în care codul poate fi analizat automat folosind embedding-uri bazate pe r-Complexitate, pentru problema de clasificare a soluțiilor din cadrul competitiilor de programare competitivă.

Rares Folea

Coordonator științifici:

Radu Iacob, Traian Rebedea

BUCUREȘTI
2022

Contents

ABSTRACT

This paper presents a solution to create dynamic code embeddings for a set of computer programs, by tailoring r-Complexity [6] functions to a wide range of metrics acquired by dynamically executing the code snippets against various input sizes. A possible application, presented in this paper, is the use of these embeddings for the problem of labeling various algorithmic solutions from competitive programming contests.

SINOPSIS

Această lucrare prezintă o soluție pentru a crea embedding-uri dinamice pentru algoritmi, prin adaptarea funcțiilor de r-Complexitate [6] la o gamă variată de metrici, dobândite în urma executiei dinamice a fragmentelor de cod pentru diferite dimensiuni ale seturilor de intrare. O posibilă aplicație, prezentată în această lucrare, este utilizarea embedding-urilor pentru problema de clasificare a soluțiilor din cadrul competițiilor de programare competitivă.

Abbreviations

AST ..... Abstract Syntax Tree

B-L ..... Bachmann-Landau

BMU ..... Best Matching Unit

BF ..... Brute Force (algorithmic technique)

BS ..... Binary Search (algorithmic technique)

CF ..... CodeForces

CPP ..... C Plus Plus (C++)

CPU ..... Central Processing Unit

CS ..... Computer Science

CV ..... Computer Vision

DC ..... Divide et Conquer (algorithmic technique)

DP ..... Dynamic Programming (algorithmic technique)

DT ..... Decision Tree

F1 ..... F-1 score (harmonic mean of precision and recall)

FC ..... Fully-Connected

I/O ..... Input/Output

JSON ..... JavaScript Object Notation

LDA ..... Linear Discriminant Analysis

LSTM ..... Long Short Term Memory

LTS ..... Long Term Support

MLP ..... Multi-Layer Perceptron

NLP ..... Natural Language Processing

NP ..... Nondeterministic Polynomial

NTFS ..... New Technology File System

OS ..... Operating System

P ..... deterministic Polynomial

PC ..... Pearson Correlation

PCA ..... Principal Component Analysis

r-C ..... r-Complexity

RAM ..... Random-Access Memory

RF ..... Random Forest

SK ..... SciKit-learn library (Python)

SP ..... Shortest Paths

SOM ..... Self-Organizing Map

XGB ..... eXtreme Gradient Boosting, also known as **XGBoost**

Chapter 1

Introduction

The goal of this research topic is to have a *better understanding of how human-written code can be automatically analyzed* and what useful characteristics can be obtained from it.

The structure of this paper consists follows an incremental presentation of the subject, following both theoretical and practical considerations. While in the first part of this work, we present the theoretical aspects of constructing algorithm embeddings based on complexity measurements, in the second part we present experimental results of the proposed models, including the methodology of evaluating the solution.

The paper begins with a dedicated **Section 1.1**, which is a preface meant to exhibit a general foreword of the problem of automatic understanding of source code. It looks at both the technical aspects of code as a system of rules to process, operate and commit information and in the overall framing it around the nonspecific universal ambiance.

Section 1.2 aims to state our motivation into investigating the problem of automatic code labeling for computer programs, with a comparison between various AI applications where the researchers are faced with multiple coexisting facts, which we have splitted based on the context in static and dynamic features. Previous related work in the field of automatic code recognition and analysing is being presented in **Section 1.3** .

Chapter 2 is dedicated to discussing various *mathematical models* for expressing the notion of complexity. **Section 2.1** presents the Classical Computational Complexity Model, defined by the family of Bachmann–Landau notations, while **Section 2.2** presents a revisited approach on an extended model of Bachmann–Landau complexities. **Section 2.3** presents a solution for automation for calculating an approximate of the associated *rComplexity* class for any given algorithm and corresponding metrics. Therefore, this chapter is a revisited approach of the work presented in [19] and [6] that provides a highly valuable theoretical initiation to r-Complexity calculus, which is the foundation of building the custom embeddings applied in the rest of the current work. For a dedicated in-depth view of this field, please refer to [19] and [6].

Chapter 3 presents how to build code-embeddings based on r-Complexity calculus and what are the requirements. Moreover, this chapter presents a simplified version that can be used to search matching r-Complexity functions.

The dataset that we have built and used in this research is widely described in **Chapter 4**. An overview of the system we built is available in **Chapter 5**. Finally, using the embeddings obtained by our system, **Chapter 6** presents how the created dynamic code embeddings can be used for the task of code classification and an analysis of their performance.

Appendix A shows the set of problems and the biggest number of corresponding crawled solutions from our training/testing dataset. **Appendix B** analyzes a sample solution from the dataset, pointing out difficulties in general automatic code analysis.

Appendix C presents a sample problem 670A, referenced in this work. **Appendix D** consists of explicitly listing the embedding generated by the system for a solution in the dataset, corresponding to problem **1366A**. **Appendix E** presents the problem 546A.

Appendix F and **Appendix G** transparently lists the full training and testing results for the task of multi label classification, using decision tree and random forest classifiers. **Appendix H** presents the results against a XGBoost classifier.

For completeness, the structure of this report consists follows an incremental presentation of the subject. The current work captures the breakdown of research spawn around 4 semesters during the Master Degree, while also pointing to the research which was done during the Bachelor Thesis, which concluded with the two articles [19] and [6]. The relevant parts of this study has been included in this article as well.

1.1 Preface

Being aware of the **meaning** of **source code** representations is a profoundly researched topic in Computer Science. While there are many fields of applications, such as computerized discoveries of bugs, detecting encryption functions in malware, and automated solutions for code generation, **automatic code labelling** is a topic that we consider worth discussing, because it is a problem that requires a **subtle understanding** on how code behaves. It can reveal solutions for many more additional problems. The problem that we approach has an inherent degree of burdensome as this requires not only profound judgment calls on the semantics of the code but also the capability of clustering algorithms based on their **commitment** and **utility**, aspects that may not be obvious only from a perfunctory analysis.

Our ambition is to make small steps towards a *tangible analysis* that can be done automatically using proposed models. We approach the problem of automatic code labeling trying to provide a general use solution, that can be easily tailored to work for other tasks as well.

Writing code seems at first glance similar to writing text in natural language. In the same

way, *reading code might be similar to reading text in natural language*. This fact led to some solutions in this field that have been inspired by similar approaches considered to be successful in natural language processing. The reason why we believe that there is a resemblance between code understanding and natural language understanding, is that both are created by humans. At the first glance, that natural language processing and code processing are different as they use completely different vocabulary and has completely different assimilation and intentions. We state that there is a similar underlying source of creation, and as a result, they are somehow similar. We can also use natural language to describe code or algorithms but the difference would be that we usually work at a higher level of abstraction. This does not make things easier, as working with human language is still an extremely difficult problem in the domain of Artificial Intelligence, as working with language processing requires a high degree of wisdom in making strong reasoning about the true semantics of the analyzed texts. A difficulty that we can immediately bring into a discussion is the multitude of spellings accepted: take for instance the **duality**[1] of accepted spelling in English of a *keyword* from the title of this research: *Automatic* **label(l)ing** *for competitive programming challenges*. *Labeled* and *labelled* are both correct spellings, as well as *Labeling* and *labelling*. The first ones are preferred in *American English*, while the others are preferable in *British English*. How should an automatic system judge these subtle differences between the two forms? Superficial answers might just suggest combining the two spelling into one that catches both meanings, and, while this might be enough to keep the majority of the features, there is an additional step to be taken by the system, and with that comes a loss of context, as knowing the allegiance of a text to the English version might be important in some conditions.

This paper will also focus on *a new approach in the field of complexities for algorithms*, by presenting a new *asymptotic computational notation* [6] that aims to provide better complexity feedback for similar computer programs, that can provide *subtle insights* even for algorithms that are part of the same conventional complexity class. The new complexity notation approaches the traditional ways of measuring the complexity of algorithms and estimating running times given the chosen architecture, in the context of dynamic infrastructure. The work included in those chapters is heavily inspired from a previous personal work [19] that introduced *new metrics for evaluating the performance and complexity of computer programs*. It also represents an enhancement that provides better sensitivity with respect to discrete analysis.

The idea of using "dynamic" embeddings has been applied in numerous cases in NLP. In [20] used this to build on exponential family embeddings to capture how the meanings of words change over time. Yet, our definitions of "dynamic" embeddings for code is slightly different. This is covered by the change over the complexity (usually we denote it by the input size), and all our measurements will be done over the axis of computing the complexity for the behaviour of certain metrics.

[1]The notion of duality is a center point of discussion in our introduction and represents the motivation for our approach.

While it seems that most of the *community work* is focused on the static code analysis, this paper aims to propose research that also takes into consideration **dynamic** analysis[2], in the sense that we desire to take a deep look on how the actual execution part of the algorithm behaves. The purpose of this is to see how the resources evolve in relation to the input size, to analyze if the increases in resource consumption (*e.g. CPU time, memory*) are proportional to input dimensions, or with the square of dimensions as well as any other dependencies that may be of interest. We believe that these properties and behaviours are quite common in the algorithms that belong to the same set. In addition, we can then look at more advanced things of computer architecture, for example, *cache-misses, scheduler timeouts*, etc. In essence, our proposal for research in this field consists of an analysis that does not only focus on static code, but a classification system that would also **dynamic** behavior analyze the algorithms.

1.2 Motivation

This section aims to present our motivation into investigating the problem of automatic code label(l)ing for computer programs.

We explained in the previous section why we believe that **dynamic** analysis will bring advantages, making analogies with legitimate examples taken from the surrounding universe. One might consider our analogy rather fabricated, with *computer vision* and *natural language processing* over-exploited, as code is a consequence of an *artificial development* of a system with a syntax and semantic to the needs of expressing computational operations. On the other hand, we support the contrary: exactly the point that code languages have been developed by humans make it *genuine* and *sincere*, that *mimics* important parts of other natural techniques, with the closest connection to natural language.

We will study in this research computer programs written in **C++**, a language that has expanded significantly over time. Modern programming language now having object-oriented, generic, and functional features. Nevertheless, C++ provides, in addition, low-level facilities such as plenty solutions for bare memory manipulation. C++ is a common choice for competitive programming challenges, as there have been developed powerful *compilers* and *optimizers* that are capable of obtaining the most out of an efficient implementation from a coding idea, providing best runtime results in class [9]. When initially launched, years ago, C++ was presented as a *high-level* language, with state-of-the-art features that were not available in previous programming languages and was providing developers with a large variation of tools to write code with, which in the end made the complementary software (compilers, debuggers, linters, automatic code recommendation systems, readability tools) extremely hard to design. Nowadays, most CS community considers C++ a *low-level* programming language that completely lacks runtime features of other dynamic programming languages (*e.g. Python, Java*)

[2]By dynamic analysis we grasp building an analysis of computer software based on data gathered from experiments performed on a real computing machine by running programs

such as implicit garbage collector, memory manipulation and platform-independence. All the (lack of) features makes this *mission* extremely interesting, as this language might be even more exciting than analysing the same solutions written in different languages.

In the *world of computing and information technology*, in terms of **static** *or/and* **dynamic** analysis, there are some patterns that arise:

1. Computer Vision: *images* **AND** *videos*, the evolution of scenes under the evolution of time.
2. Natural Language Processing: *text* **AND** *sound*, capturing different intonations, longer shorter pauses in speaking.
3. **Code analysis**: *static* **AND** *dynamic* analysis. Looking at static written code against run dynamically the code.

This nation of **duality**, applied in the generic problem of *computer vision* can be expressed as having an algorithmic that can interfere an understanding similar to our knowledge we obtain during the act of seeing. Serious hard questions arise from this goal, such as finding the true meaning of what vision is. Proven to be a question with deep philosophical consequences, it has been widely studied with ambitious goals of formulating a coherent theory on what vision is and how it can be automatically inherited by computer using dedicated software built on the clear algorithmic solution to the vision problem. The reality has proven to be far more difficult and complex than it has been initially thought as yet, the problem of vision is far away from being solved. Modern approach offers decent results for systems that use advanced machine learning techniques on data that are similar to the environment they have been trained on. However, the view of the worlds might be totally different and using systems that have been trained on a particular context might under-perform on different scenarios. Moreover, best computer vision systems decided to switch from a traditional approach on analyzing bare images individually, having an independent-frame analysis, into analysing videos as a whole, with time causality dependencies between frames. This reaches our point: the field of computer vision managed to evolve and obtain better systems by taking the **advantages of duality**; for such systems, *images* **AND** *videos* are used together for enriching the understanding of vision, linking the evolution of scenes with the evolution of time. Just by taking individual images and having a solo deep analysis is valuable, but by doing so, the system loses the context[3] which is undoubtedly valuable for the general awareness.

Switching our attention to the problem to the *natural language recognition* by automated computers systems, our suggestion is to imagine how humans manage to find *insights* from simple symbols to provoke neural incentives in the human brain to process the information behind. It is a wonder that we have been endorsed with this fantastic abilities, and even though we don't fully understand the genuine mechanics behind how this process happens, the intuition tells us that language represents more than vague symbols. In casual conversations, we are *empowered* with multiple censorial features that helps the whole process excel:

[3]Which we will further refer as dynamic context.

when hearing a message in a given natural language, we are capable of capturing different intonations, longer shorter pauses in speaking, and even emotions. However, reading text reduces the *sound* advantages of the spoken language, as this gets completely lost[4]. Theories that suggest that the human brain does more than just reading arises, and we couldn't agree more: there must be something deeper that brings meaning out of written language: to support this, we would like to present a little example from *literature*. *The Raven* is a narrative poem written by *Edgar Allan Poe*, a remarkable American writer[5]. Published initially in 1845, the poem is always noted for its *musicality*, *stylised expression*, and *mystical atmosphere*. It tells of the enigmatic visit of a talking raven to a distressed lover, tracing the man's gradual descent into madness. The lover is lamenting the loss of his love, Lenore, also known as a student. Sitting on a Pallas bust, with his frequent repetition of the word **Nevermore**, the raven seems to annoy the protagonist further. With eleven occurrences of the Nevermore in a short poem, our understanding of the raven goes far ahead of the trivial *metaphor* that the raven is able to speak.

We are enriched with the ability of seeing more than simple written language, as we associate meaning to that. When reading this, we can imagine the tonality of bird, and associate that with a feeling of grave. By doing so, we exceeded simple text analysis, we advanced to the next level: **duality** between *text* **AND** *sound*, capturing far more information that from the plain symbols. Getting back to the problem of building software for processing the natural language, what a major achievement would it be if we would be able to have all the help computations that the human brain does, that are consequences to *analyzing the sound behind the text*. In this research, our ambition is to obtain and apply the **duality** for **software** itself and more specifically, for computer programs written for solving programming challenges.

1.3 Related work

In this section we will present the *state-of-the-art* researches in the domain of automatic code recognition[6] and analysing. We will use just some of the below *techniques* in building our model, but the general ideas behind brings value to our understanding of how we can perform operations automatically on *human-written* code.

Generating embeddings[7] is a useful widely applied technique in the field of natural language

[4]The loss may be avoided with advanced systems that offers solutions for emulating the sound and melody of the plain text.

[5]We will get later to our point on how examples from literature can help us develop better computer algorithms and models for solving our particular problem

[6]Even though the title of our research indicates that our models are only aiming for automatic labelling of computer programs, we see value in having understanding beyond that. Finding similar solutions that worked for similar problem might provide us better tooling for operating on this particular problem and this enlargement over the view of the problem might open paths in other directions as well.

[7]Embeddings are generally useful, as they describes a mapping to a continuous numerical vector of a discrete variable. Embeddings are generally useful due to the property that the representation space of discrete

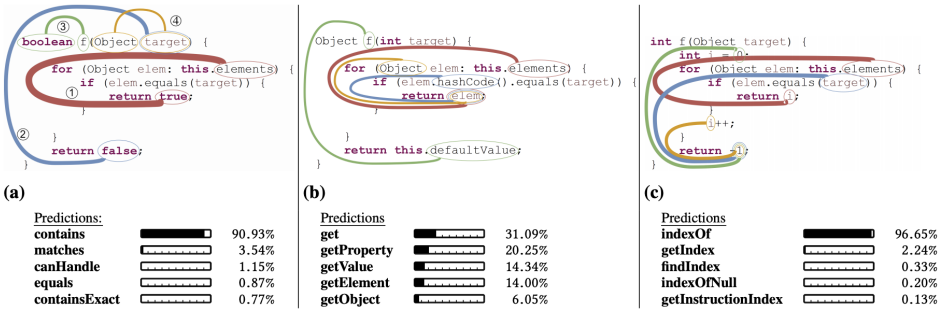


Figure 1: Some interesting results outputted by the algorithm, for three Java methods, for which the code2vec model catches the subtle differences between similar yet different methods. Image credits belong to [2].

processing (see *word2vec* model [16, 17]), by representing words or phrases as a single fixed-length vector, which can be used to predict semantic properties of the analyzed fragments. One[8] interesting approach to this technique has been applied in **code2vec** [2], a research paper that provides a learning technique for building representations of code.

The work presents a *neural model* for representing snippets of code as *continuous distributed vectors*. The chosen nomenclature for this vectors representations is code embeddings, fixed-length code vectors, that can be used to predict semantic properties of the code snippet. To this end, code in its abstract syntax tree [24] is first decomposed into a set of paths. The network then learns each path's atomic representation while simultaneously learning how to aggregate a collection of them.

The **code2vec** model has been initially used to propose an automatic system for proposing function names, but the core algorithm creates vectors called code embeddings that can be generally used and it is language agnostic. Even though the original research was applied on Java code, studies have shown solutions on how to extend this solutions to other languages, such as Python, C, C++ or C#.

The same authors later proposed a new model, **code2seq** [1], with the goal of generating sequences from structured representations of code. As the name indicates, it is an algorithm from the *Seq2seq* family: approaches to machine learning used for language processing. Its core idea is to turn one sequence into another sequence. Similar to code2vec, the main concept is to sample paths from the *Abstract Syntax Tree* of a code snippet, encode these paths with an LSTM[9], and attend to them while generating the target sequence. While this approach outclassed the code2vec model results, it is harder to be applied in the problem of auto labelling as the model is particularized for the specific tasks described in the paper: code summarizing and code captioning.

variables can be reduced while the properties of the variables in the space transferred can be preserved.

[8] Also check [12] for another solution on understanding by building up embeddings.

[9] A *long short-term memory* (**LSTM**)[7] is an recurrent neural network with feedback connections that is used in the field of deep learning.

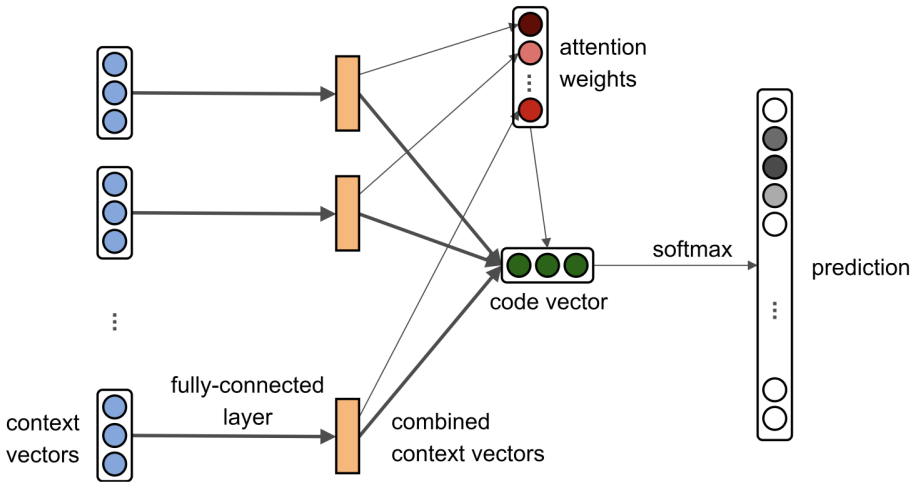


Figure 2: The architecture of code2vec path-attention network. A full-connected layer learns to combine the path embedding between two leaves from the *abstract syntax tree* together with the token embeddings of the leaves, as attention weights are learned using the combined context vectors, and used to compute a code vector. The resulting code vector is used in making the prediction. Description and image credits belong to [2].

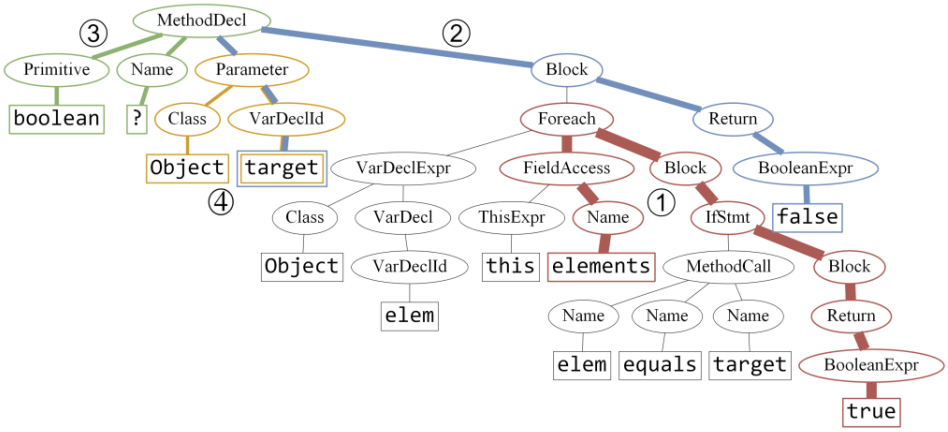


Figure 3: Sample construction of a *Abstract Syntax Tree* for a code-snippet implementing a classic contains Java-like method. The attentions provided in the code2vec paper were: red *(path 1)*: 0.23, blue *(path 2)*: 0.14, green *(path 3)*: 0.09, orange *(path 4)*: 0.07. Image credits and results belong to [2].

If these two approaches are **AST**-based and operate on the source code (by only following anything else than syntax patterns, without looking into variables at a contextual level), there have been various studies that aim to look on code at a *lower level*, inspecting the generated, compiled, *binary code*. Even though at first glance this approach appears to be more difficult due to the contraction of language **expressiveness**[10] and the **decrease** in **readability**, some models appear to perform pretty well on certain tasks. We present a simple example for proving this *liability* argument: consider the problem of *computing the square of an integer*. A simple **C**-like implementation of a function that would solve this trivial problem would be:

```
int square(int num) {
    return num * num;
}
```

The resulting assembly snippet after compiling this on a *x86-64* architecture would be similar to the following:

```
square(int):
    push    rbp
    mov     rbp, rsp
    mov     DWORD PTR [rbp-4], edi
    mov     eax, DWORD PTR [rbp-4]
    imul    eax, eax
    pop     rbp
    ret
```

The similarity of the first snippet written in **C** with respect to the natural language is obviously much higher than the similarity of the *x86-assembly*, which is closer tied to the machine language of the computing machine. All the interesting logic is present in the $imul$ *instruction*, with the rest of the code representing noise caused by architectural details of the instruction set and programming conventions.

Moreover, these models can provide solutions to problems where the source code is *unavailable* (*e.g. plagiarism detection, malware detection, vulnerability search*) or there are more that one languages for the analyzed scenario[11]. Such results in literature give us the confidence that generating embeddings from assembly languages is valuable and we may be able to use these techniques in order to increase the overall performances of our predicting labels models. This

[10] At this point we can open a completely new discussion on why operating with a higher level-language text is easier than operating with binary files, as we should begin with the reasonable naming of the two: a high-level programming language is a programming language that is highly abstracted from the computer's architectural details. This languages are using natural language elements, in order to be simpler to use. They also automate some important computer system areas such as memory management, or even completely hide the primitives, making the process of software creation easier and more comprehensible than when using a lower-level language. What a high-level programming language is depends on multiple things. If, for instance, you would have ask this question 50 years ago, an answer like Fortran would have been dominating, while as of today, Python, Go or Java are the most common examples of high-level programming languages provide a much larger abstraction over the physical machinery.

[11] Parts of software suites that are written in different programming languages using completely different techniques.

can be achieved simply by having an *ensemble* model whose decision sub model also includes the analysis of generated assembly code.

In [25] it is presented an approach based on a deep neural network to produce embeddings for a binary function. It uses a graph embedding network that was adapted from Structure2vec [4], trained specifically for the problem of differentiating the similarity between two inputs.

Chapter 2

Complexities

The following chapter is dedicated to discussing various *mathematical models* for expressing the notion of complexity. Starting from the classical Big-O Complexity Model, we will transition to a custom complexity model proposed in a previous personal work [19], [6] that introduced *new metrics for evaluating the performance and complexity of computer programs*. Complexities are *essential* in our contribution for solving the problem of automatic code labelling in building the dynamic model that aims to fit a specific algorithm in its belonging class. Our research wants to provide a full understanding on how it was build, providing a translucent view on manufacturing details behind and on this philosophy we will invest a vast amount of resources in describing what exactly is being calculated by our system.

2.1 The Classical Complexity Model and the family of B-L notations

For explaining the asymptotic behavior of the complexity of an algorithm characterized by a function, the following notations and names are used:

1. Let f be the characterized function of the algorithm, and let the function be defined on: $f : \mathbb{N} \longrightarrow \mathbb{R}$.
2. The set of all complexity calculus $\mathcal{F} = \{f : \mathbb{N} \longrightarrow \mathbb{R}\}$.
3. Assume that $n, n_0 \in \mathbb{N}$.
4. Consider an arbitrary complexity function $g \in \mathcal{F}$.

The following function classes can be therefore defined for any function $g : \mathbb{N} \longrightarrow \mathbb{R}$ that describes a convergent sequence or a divergent sequence with a limit that tends to infinity:

Definition 2.1.1. Big Theta: *This set defines the group of mathematical functions similar in magnitude with $g(n)$ in the study of asymptotic behavior. A set-based description of this*

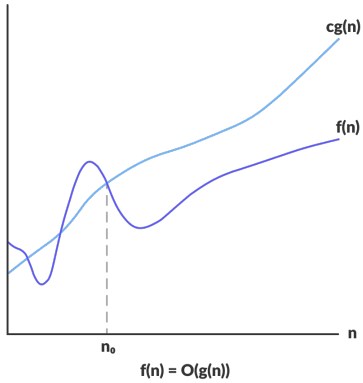


Figure 4: A data visualisation of a function f that is bounded by the Big-O complexity class of the function g. n_0 represents a point starting from which for any $n \geq n_0$, $f(n) \leq c \cdot g(n)$.

group can be expressed as:

$$\Theta(g(n)) = \{f \in \mathcal{F} \mid \exists c_1, c_2 \in \mathbb{R}_+^*, \exists n_0 \in \mathbb{N}^* \ s.t. \ \ c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \ , \ \forall n \geq n_0\}$$

Definition 2.1.2. Big O: *This set defines the group of mathematical functions that are known to have a similar or lower asymptotic performance in comparison with $g(n)$. The set of such functions is defined as it follows:*

$$\mathcal{O}(g(n)) = \{f \in \mathcal{F} \mid \exists c \in \mathbb{R}_+^*, \exists n_0 \in \mathbb{N}^* \ s.t. \ f(n) \leq c \cdot g(n), \ \forall n \geq n_0\}$$

Definition 2.1.3. Big Omega: *This set defines the group of mathematical functions that are known to have a similar or higher asymptotic performance in comparison with $g(n)$. The set of all function is defined as:*

$$\Omega(g(n)) = \{f \in \mathcal{F} \mid \exists c \in \mathbb{R}_+^*, \exists n_0 \in \mathbb{N}^* \ s.t. \ f(n) \geq c \cdot g(n), \ \forall n \geq n_0\}$$

Lemma 2.1.1. *In order to establish a Big-Omega acceptance, a sufficient condition is one of the following:*

(i)
$$\lim_{x \to \infty} \frac{f(x)}{g(x)} = -\infty \Rightarrow f \in \Omega(g(n))$$

(ii)
$$\lim_{x \to \infty} \frac{f(x)}{g(x)} = \infty \Rightarrow f \in \Omega(g(n))$$

(iii)
$$\lim_{x \to \infty} \frac{f(x)}{g(x)} = C \in (-\infty, 0) \cup (0, \infty) \Rightarrow f \in \Omega(g(n))$$

2.2 A refined complexity calculus model: r-Complexity

This section contains a reviewed presentation of a complexity model proposed in a previous personal work [19], [6] that introduced *new metrics for evaluating the performance and complexity of computer programs*.

2.2.1 Introduction and Motivation

Similar to the conventional asymptotic notations proposed in the literature, *rComplexity* is expressed as a function $f : \mathbb{N} \longrightarrow \mathbb{R}$, where the function is characterized by the size of the input, while the evaluated value $f(n)$, for a given input size n, represents the amount of resources needed in order to compute the result. This proposed [6] calculus model intends to generate new asymptotic notations that provide richer complexity analysis for comparable algorithms, delivering subtle insights also for algorithms that belonged to the same traditional complexity class. $\Theta(g(n))$, denoted by an arbitrary function $g : \mathbb{N} \longrightarrow \mathbb{R}$, in the *B-L* definitions.

While the classical complexity calculus model is a long-established, verified metric of evaluating an algorithm's performance and it is a valuable measurement in estimating feasibility of computing a considerable algorithm, the model has a few shortfalls in making discrepancy between similar algorithms with two similar complexity functions, $v, w : \mathbb{N} \longrightarrow \mathbb{R}$, such that $v(n), w(n) \in \Theta(g(n))$. In order to highlight the lack of distinction, suppose two algorithms $Alg1$ and $Alg2$ that solve exactly the same problem[1], with the following complexity functions: $f_1, f_2 : \mathbb{N} \longrightarrow \mathbb{R}$ where $f_1 = x \cdot f_2$, with $x \in \mathbb{R}_+,\ x > 1$. If

$$f_2 \in \Theta(g(n)) \Rightarrow \exists c_1, c_2 \in \mathbb{R}_+^*, \exists n_0 \in \mathbb{N}^*\ s.t.\ \ c_1 \cdot g(n) \leq f_2(n) \leq c_2 \cdot g(n)\ ,\ \forall n \geq n_0$$

Therefore, $f_1 \in \Theta(g(n))$, as there exists $c_1^{'}, c_2^{'} \in \mathbb{R}_+^*, n_0^{'} \in \mathbb{N}^*$ such that $c_1^{'} = x \cdot c_1$, $c_2^{'} = x \cdot c_2$ and $n_0^{'} = n_0$ and $\forall n \geq n_0^{'} :\ c_1^{'} \cdot g(n) \leq f_1(n) \leq c_2^{'} \cdot g(n)$.

rComplexity calculus aims to solve this issue by taking into deep analysis the preeminent constants that can state major improvements in an algorithm's complexity and can have tremendous effect over total execution time [6].

[1] Remark that even if $f_1 > f_2$, both complexity functions are part of the same complexity class. This observation implies that two algorithms, whose complexity functions can differ by a constant, even as high as 2022^{2022}, are part of the same complexity class, even if the actual run-time might differ by over tens of thousands of orders of magnitude. For comparison, only a 30 magnitude order between the two complexity functions $f_1 = 10^{30} \cdot f_2$, signify that if for a given input n, if $Alg2$ ends execution in 1 *attosecond* (10^{-9} part of a nanosecond), then $Alg1$ is expected to end execution in about 3 *millenniums*. Despite of the colossal difference in time, classical complexity model is not perceptive between algorithms whose complexity functions differs only through constants.

2.2.2 rComplexity definitions

The following notations and names will be used for describing the asymptotic behavior of a algorithm's complexity characterized by a function, $f : \mathbb{N} \longrightarrow \mathbb{R}$.
We define the set of all complexity calculus $\mathcal{F} = \{f : \mathbb{N} \longrightarrow \mathbb{R}\}$
Assume that $n, n_0 \in \mathbb{N}$. Also, we will consider an arbitrary complexity function $g \in \mathcal{F}$. Acknowledge the following notations $\forall r \neq 0$:

Definition 2.2.1. Big r-Theta: *This set defines the group of mathematical functions similar in magnitude with $g(n)$ in the study of asymptotic behavior. A set-based description of this group can be expressed as:*

$$\Theta_r(g(n)) = \{f \in \mathcal{F} \mid \forall c_1, c_2 \in \mathbb{R}_+^* \ s.t. c_1 < r < c_2, \exists n_0 \in \mathbb{N}^* \\ s.t. \ \ c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \ , \ \forall n \geq n_0\}$$

Definition 2.2.2. Big r-O: *This set defines the group of mathematical functions that are known to have a similar or lower asymptotic performance in comparison with $g(n)$. The set of such functions is defined as it follows:*

$$\mathcal{O}_r(g(n)) = \{f \in \mathcal{F} \mid \forall c \in \mathbb{R}_+^* \ s.t. \ r < c, \exists n_0 \in \mathbb{N}^* \ s.t. \ f(n) \leq c \cdot g(n), \ \forall n \geq n_0\}$$

Definition 2.2.3. Big r-Omega: *This set defines the group of mathematical functions that are known to have a similar or higher asymptotic performance in comparison with $g(n)$. The set of all function is defined as:*

$$\Omega_r(g(n)) = \{f \in \mathcal{F} \mid \forall c \in \mathbb{R}_+^* \ s.t. \ c < r, \exists n_0 \in \mathbb{N}^* \ s.t. \ f(n) \geq c \cdot g(n), \ \forall n \geq n_0\}$$

2.2.3 Interdependence properties

An interesting property of the **Big r-Theta**, **Big r-O** and **Big r-Omega** classes is the simple technique of conversion between various values for the rs parameters. The following results arise:

Theorem 2.2.1. Big r-Theta conversion:

$$f \in \Theta_r(g) \Rightarrow f \in \Theta_q\left(\frac{q}{r} \cdot g\right) \ \forall r, q \in \mathbb{R}_+$$

Proof. $f \in \Theta_r(g(n)) \Rightarrow \forall c_1, c_2 \in \mathbb{R}_+^* \ s.t. c_1 < r < c_2, \exists n_0 \in \mathbb{N}^*$
$s.t. \ \ c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \ , \ \forall n \geq n_0$
Therefore, $\forall c_1, c_2 \in \mathbb{R}_+^* \ s.t. c_1 < q < c_2, \exists n_0 \in \mathbb{N}^*$
$s.t. \ \ \frac{r}{q} \cdot c_1 \cdot g(n) \leq f(n) \leq \frac{r}{q} \cdot c_2 \cdot g(n) \ , \ \forall n \geq n_0.$
Thus, $f \in \Theta_q\left(\frac{q}{r} \cdot g\right)$. ■

Another interesting result is obtained by multiplying the last equation by $\frac{q}{r}$:
$\forall c_1, c_2 \in \mathbb{R}_+^* \ s.t. c_1 < q < c_2, \exists n_0' = n_0 \in \mathbb{N}^*$
$s.t. \ \ c_1 \cdot g(n) \leq \frac{q}{r} \cdot \frac{r}{q} \cdot f(n) \leq c_2 \cdot g(n) \ , \ \forall n \geq n_0$

Corollary 2.2.1.1.

$$\frac{q}{r} \cdot f \in \Theta_q(g)$$

.

Theorem 2.2.2. Big r-O Conversion:

$$f \in \mathcal{O}_r(g) \Rightarrow f \in \mathcal{O}_q\left(\frac{q}{r} \cdot g\right) \ \forall r, q \in \mathbb{R}_+$$

Proof. Is similar with the proof for Big r-Theta, with the inequalities becoming $f(n) \leq c \cdot g(n)$. ■

Corollary 2.2.2.1. *The following conversion relationship arises:*

$$f \in \mathcal{O}_r(g) \Rightarrow \frac{q}{r} \cdot f \in \mathcal{O}_q(g) \ \forall r, q \in \mathbb{R}_+$$

Theorem 2.2.3. Big r-Omega Conversion:

$$f \in \Omega_r(g) \Rightarrow f \in \Omega_q\left(\frac{q}{r} \cdot g\right) \ \forall r, q \in \mathbb{R}_+$$

Proof. Is similar with the proof for Big r-Theta, with the inequalities becoming $f(n) \geq c \cdot g(n)$. ■

Corollary 2.2.3.1. *The following conversion relationship arises:*

$$f \in \Omega_r(g) \Rightarrow \frac{q}{r} \cdot f \in \Omega_q(g) \ \forall r, q \in \mathbb{R}_+$$

There are some interesting relationship results regarding the connection between $rComplexity$ functions and the correspondent class in the $Bachmann - Landau$ notations. For Big notations $(\Theta, \mathcal{O}, \Omega)$, we present further some results:

Theorem 2.2.4. *Relationship between Big r-Theta and Big Theta:*

$$f \in \Theta_r(g) \Rightarrow f \in \Theta(g)$$

$$f \in \Theta(g) \Rightarrow \exists r \in \mathbb{R}_+ \ f \in \Theta_r(g)$$

Theorem 2.2.5. *Relationship between Big r-O and Big O:*

$$f \in \mathcal{O}_r(g) \Rightarrow f \in \mathcal{O}(g)$$

$$f \in \mathcal{O}(g) \Rightarrow \exists r \in \mathbb{R}_+ \ f \in \mathcal{O}r(g)$$

Theorem 2.2.6. *Relationship between Big r-Omega and Big Omega:*

$$f \in \Omega_r(g) \Rightarrow f \in \Omega(g)$$

$$f \in \Omega(g) \Rightarrow \exists r \in \mathbb{R}_+ \ f \in \Omega_r(g)$$

2.3 Automatic estimation of rComplexity

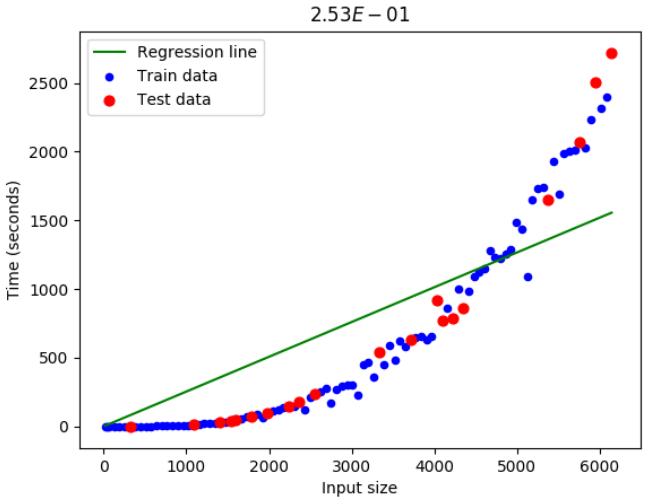
This section aims to present a solution for automation for calculating an approximate of the associated rComplexity class for any given algorithm. The prerequisites for this method implies a technique for obtaining relevant metric-specific details for diversified input dimensions. For instance, if time is the monitored metric, there must exist a collection of pertinent data linking the correspondence between input size and the total execution time for the designated input size.

2.3.1 Estimation for algorithms with known B-L Complexity

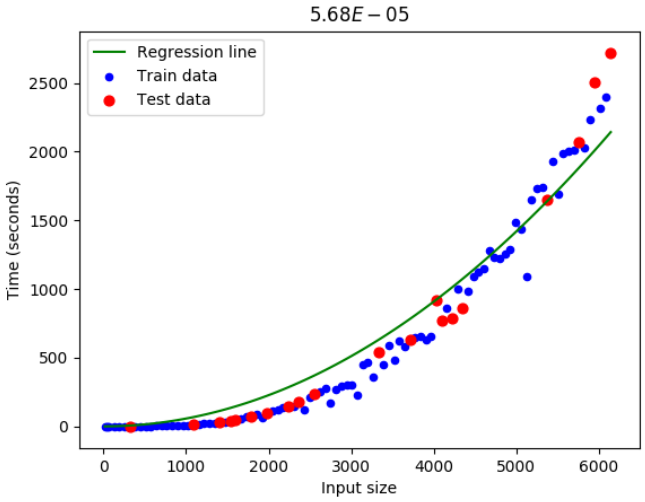
Estimating $f(n)$, the associated rComplexity class for an algorithm with established B-L Complexity $g(n)$, consists in the process of tailoring an suitable constant c, such that $f \approx \Theta_1(c \cdot g)$. We propose solving this problem with linear regression, with adjusted entry values, when the B-L relationship between the $inputSize$ and the metric is known. In order to adjust the learning set to a more knowledgeable set, we can extract new features and replace all the ($inputSize$, $metricValue$) pairs with ($g(inputSize)$, $metricValue$), where g is the known B-L Complexity function.

Training a linear regression model on this dataset would result in model with a linear equation to observed data, similar with the one below(a). Changes in the original dataset can enhance fitting results for the regression.

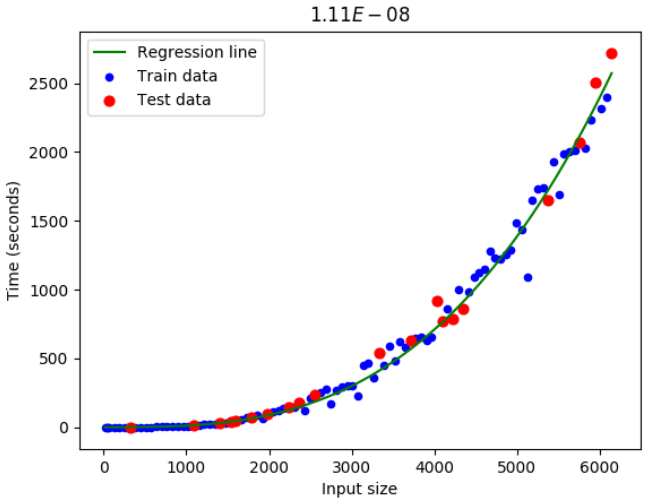
As an intuition (due to the associated complexity function $O(n^3)$ in the B-L Complexity model), the natural fit was obtained when using $g(n) = n^3$ with consideration to generalization. If we choose much much bigger degree polynomial transformations, we may obtain better results on this datasets, but the models are becoming subject to overfit.



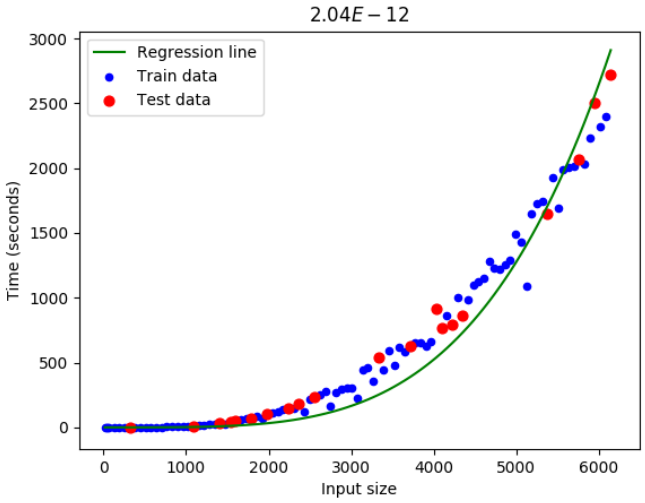
(a) Linear Regression Model trained with features obtained using the transformation $g(n) = n^1$



(b) Linear Regression Model trained with features obtained using the transformation $g(n) = n^2$



(c) Linear Regression Model trained with features obtained using the transformation $g(n) = n^3$



(d) Linear Regression Model trained with features obtained using the transformation $g(n) = n^4$

Figure 5: Various prediction boundaries based on accommodated training dataset using multiple relations g. Training data are obtained for different input size for a naive matrix multiplication algorithm in $\mathbb{O}(n^3)$. Credits to [6].

2.3.2 Estimation for algorithms with unknown B-L Complexity

Estimation for algorithms with unknown B-L Complexity becomes a lot more difficult as there are numerous possible candidates for a matching complexity function.

A general polynomial Performance model normal form is presented in Chapter 2.1 Automatic Empirical Performance Modeling of Parallel Programs [3]. An enhanced model for complexity functions should contain also an exponential behavior, which is often seen as a synergy between

NP-Hard problems. Thus, we propose the following general expression:

$$f(n) = \sum_{t=1}^{y} \sum_{k=1}^{x} c_k \cdot n^{p_k} \cdot log_{l_k}^{j_k}(n) \cdot e_t^n \cdot \Gamma(n)^{g_k}$$

This representation is, of course, not exhaustive, but it works in most practical schemes. An intuitive motivation is a consequence of how most computer algorithms are designed. [3]

For the purpose of this paper, we will use a set of common time complexities as described in [23] and [3].

Notable classes are the logarithmic time[2], polynomial time[3] and exponential time[4], as presented in **Table 2.1**.

Name	**Running Time**	**Examples**
constant time	$O(1)$	Common arithmetical operations.
log-logarithmic	$O(log\ log(n))$	Amortized time per operation using a bounded priority queue.
logarithmic time	$O(log(n))$	Binary search.
fractional power	$O(n^c),\ 0 < c < 1$	Searching in a KD-tree.
linear time	$O(n)$	Finding the smallest number in an unsorted array
linearithmic time	$O(n \cdot log(n))$	Fastest possible comparison sort.
quadratic time	$O(n^2)$	Bubble, Insertion sort
cubic time	$O(n^3)$	Naive multiplication of two n x n matrices.
polynomial time	$poly(n)$	Any linear, quadratic, cubic example
exponential time (with linear exponent)	$2^{O(n)}$	Solving the traveling salesman problem using dynamic programming.
exponential time	$2^{poly(n)}$	Solving matrix chain multiplication via brute-force search.
factorial time	$O(n!)$	Solving the Travelling salesman problem via brute-force search .

Table 2.1: Some common time complexities for various well-known algorithms, adapted from [23].

[2]In computational complexity theory, **DLOGTIME** is the complexity class of all computational problems solvable in a logarithmic amount of computation time on a deterministic Turing machine.[23]

[3]In computational complexity theory, **P**, also known as **PTIME**, is a fundamental complexity class. It contains all decision problems that can be solved by a deterministic Turing machine using a polynomial amount of computation time, or polynomial time.[23]

[4]In computational complexity theory, the complexity class **EXPTIME** is the set of all decision problems that are solvable by a deterministic Turing machine in exponential time.[23]

Chapter 3

Dynamic code embeddings based on r-Complexity

3.1 Theoretical foundation

As shown in [6] and in **Chapter 2.2**, r-Complexity is a refined complexity calculus model that provides a new asymptotic notation that offers better complexity feedback for similar programs than the traditional Bachmann-Landau notation, providing subtle insights even for algorithms that are part of the same conventional complexity class. The big **r-Theta** notation proves to be useful also in creating Dynamic code embeddings. The idea behind these embeddings is simple: try to automatically provide estimations, for various metrics, the r-Theta class for the analyzed algorithm (usually with unknown Bachmann–Landau Complexity).

A generic solution to providing automatic estimation of r-Complexity has been analyzed in **Section 2.3.2**. However, we will need to instantiate the model thereby presented to analyze tangible metrics with respect to the input size.

In this research, we will attempt to fit a simplified version of the generic expression as a Big r-Theta function, described generic by one of the following function:

$$\begin{cases} r \cdot log_2^p log_2(n) + X \\ r \cdot log_2^p(n) + X \\ r \cdot p^n + X, p < 1 \\ r \cdot n^p + X \\ r \cdot p^n + X, p > 1 \\ r \cdot \Gamma(n) + X \end{cases}.$$

As the search space of such functions is infinite, in practice it is not feasible to perform an exhaustive search of all combinations.

3.2 Discretizing the search space

The search space for regressor functions is not feasible to be exhaustively searched during for automatic computation, as there are infinitely many regressors that can be analyzed. To overcome this problem, our approach is that we can avoid performing the continuous space search, and instead we can obtain similar results only by sampling a number of highly relevant configurations, that are relevant for algorithms in general. The functions we decided to search towards are:

$$\begin{cases} r \cdot log_2^p log_2(n) + X, p \in \{0, 1, 2, 3\} \\ r \cdot log_2^p(n) + X, p \in \{0, 1, 2, ..., 10\} \\ r \cdot p^n + X, p < 1, p \in \{0.1, 0.2, ..., 0.9\} \\ r \cdot n^p + X, p \in \{1, 1.3, 1.5, 1.7, 2, 2.5, 2.7, 3, 3.5, 4, 4.5, 5, 5.5, 6, 7, 8, 9, 10\} \\ r \cdot p^n + X, p > 1, p \in \{1.5, 2, 2.5, 3, 3.5, 4, 5\} \\ r \cdot \Gamma(n) + X \end{cases}.$$

Each fitting of a Big r-Theta on a metric has as result a quadruple:

`(FEATURE_TYPE, FEATURE_CONFIG, INTERCEPT, R-VAL)`

where:

1. `FEATURE_TYPE` has one of the following values[1]:
   (a) `LOGLOG_POLYNOMIAL`,
   (b) `LOG_POLYNOMIAL`,
   (c) `FRACTIONAL_POWER`,
   (d) `POLYNOMIAL`,
   (e) `POWER`,
   (f) `FACTORIAL`,
2. `FEATURE_CONFIG` is defined[2] by a value $n \in \mathbb{R}_+$, such that the generic Big r-Theta respects the above `FEATURE_TYPE`.
3. `INTERCEPT` is the expected value of the complexity function when the input size is null.
4. `R-VAL` is the value of r, such that the defined g complexity class includes f, which is the real complexity function of the algorithm: $f \in \Theta_r(g(n))$

Multiple fits[3] of a Big r-Theta complexity function on multiple metrics creates a chaining of quadruples, that can create a code embedding, which can be further analyzed for solving multiple tasks. The resulting analyzed regressors that we decided to monitor, corresponding to the fitting of r-Complexity functions are summarized in **Table 3.1**.

[1]The model is generic and other values can be used as well, yet these are the most relevant values that we have used in our research.

[2]In our research, we have searched only a small discrete set of values for n, described earlier in this section.

[3]We aim to create realistic approximations of the real Big r-Theta complexity, using regression models.

FEATURE_TYPE	FEATURE_CONFIG	FEATURE_TYPE	FEATURE_CONFIG
LOGLOG_POLYNOMIAL	0	POLYNOMIAL	1
	1		1.3
	2		1.5
	3		1.7
LOG_POLYNOMIAL	0		2
	1		2.5
	2		2.7
	3		3
	4		3.5
	5		4
	6		4.5
	7		5
	8		5.5
	9		6
	10		7
FRACTIONAL_POWER	0.1		8
	0.2		9
	0.3		10
	0.4	POWER	1.5
	0.5		2
	0.6		2.5
	0.7		3
	0.8		3.5
	0.9		4
FACTORIAL	1		5

Table 3.1: The setup of the analyzed regressors for discretizing the search space.

3.3 Particularization for algorithms metrics

Our embeddings can be built using any metrics that are being collected during the execution of the algorithm. The more distinguishable they are between algorithms, the more knowledge will the code embedding be able to capture and the better the performance will be when performing classification.

As a proposal, this paper provides a large set of predefined computed metrics for analyzed dataset, including but not limited to the following metrics:

- branch-misses [4]
- branches
- context-switches [5]
- cpu-migrations
- cycles
- instructions
- page-faults [6]
- stalled-cycles-frontend
- task-clock
- Average resident set size
- Average shared text size
- Average stack size
- Average total size
- File system inputs
- File system outputs
- Involuntary context switches
- Major (requiring I/O) page faults
- Maximum resident set size
- Minor (reclaiming a frame) page faults
- Signals delivered
- Socket messages received
- Socket messages sent
- Swaps [7]
- System time
- User time
- Voluntary context switches

For different configuration of the function described above, we aim to fit Regressions to best fit the dynamic variations of the acquired metrics. The description of how we acquired in practice this metrics is available in **Section 4.3**.

A sample embedding generated by our system can be consulted in **Appendix D**.

[4] [5] presents the limits of Indirect Branch Prediction and an overview of Branch Prediction in general.

[5] Measuring the indirect cost of context switch is a challenging problem [15].

[6] Page-faults raise when a process accesses a memory page without proper preparation.

[7] Swapping is a memory management technique that allows any process to be temporarily switched from main memory to a secondary memory, freeing up space in the main memory that can be used for other tasks.

Chapter 4

Dataset

The set of data required for this project is considerable and due to its nature is divided in multiple parts. This chapter will individually present each piece of the puzzle that together make up the entire required input to run this project and the created set of data. In this research, we have used as a starting point **AlgoLabel** [11]: *a dataset for Multi-Label Classification of Algorithmic Challenges* developed specifically for tasks that consists of labelling the algorithmic solution for programming challenges. From this dataset, we have used the labels for some Codeforces[1] problems, that classify an algorithm in a given class. However, the total number of available solutions for Codeforces problems was low in this dataset, with only little over a thousand solutions. We wanted to *enrich* this dataset with many more solutions, by leveraging the open-sources solutions from public repositories. The development of this dataset and pre-processing tools can be found on GitHub[2] NLCP@AlgoLabel.

To summarize, the data-sources for the software suite required to run consisting of the following repositories for data storing:

1. **TheInputsCodeforces**: Generator functions and the generated synthetic inputs.
2. **TheOutputsCodeforces**: The performance analysis repository and the obtained metrics from the dynamic analysis.
3. **TheCrawlCodeforces**: The dataset with the code sources collected from open-source project that solves various problems from the Codeforces platform.

4.1 C++ Sources

We called this operation **TheCodeforcesCrawled**, that is, an automatic system for building software solutions for automatic discovery of code solutions implemented and *open-sourced* by authors from all over the world and gathering the results in an unique centralized environment

[1] Codeforces is a platform that hosts online competitive programming contests. https://codeforces.com
[2] Last accessed on 31/01/21: https://github.com/NLCP/AlgoLabel

OS	Total number of sources	Total number of assembly language files generated	Total number of compiled sources	Percentage of usable sources(%)
Ubuntu	140905	135701	134426	95.4%
macOS	140905	112150	110922	78.72%

Table 4.1: A report on the absolute values on which we have created the dataset during the TheCrawlCodeforces project.

OS	Size (on disk, 4K block size) of the total number of sources	Size (on disk, 4K block size) of assembly language files generated	Size (on disk, 4K block size) of compiled sources
Ubuntu	0.18 GB (0.65 GB)	8.59 GB (9.01 GB)	4.14 GB (4.52 GB)
macOS	0.18 GB (0.65 GB)	9.23 GB (9.67 GB)	6.81 GB (7.13 GB)

Table 4.2: The size of the dataset obtained during the TheCrawlCodeforces operation.

that can be then further used by our models.

With this approach we were able to expand the previous dataset by many orders of magnitude. During this research, we have obtained over 65GB of raw code data from internet containing solutions for problems published in Codeforces contests. Our software did an automatic raw cleanup in that amount of data resulting in raw **173076** sources.

After additional filtering and cleanup steps both automatic and manual similar to the ones described in[11], the total number of code sources that compiled successfully for CF[3] problems reached a peak of 140905 unique sources. With further processing, we have successfully managed to compile a total of **134426**[4] solutions to thousands of CF problems. All the analyzed code was written in C++. This process continues with pipelines that automatically compile existing CPP files and stores the resulted assembly code as well as the executable (platform-dependent).

This process took a fair amount[5] of time, as analysis and prepossessing have been done serially due to the limitations of available hardware and storage systems. We have also decided to try to compile the same dataset for multiple minor versions of Intel x86 architectures on two different operating systems[6]: *MacOS Catalina version 10.15.7 (19H15)* and *Ubuntu Focal Fossa 20.04.1 LTS* with *Linux Kernel version 5.4*.

[3](abbreviation): CodeForces - https://codeforces.com/

[4]The number may slightly vary based on the chosen host operating system.

[5]Compilation and filtering (executed serially) took around 5 computing days, for each OS independently.

[6]The compiled datasets are available online on two different custom-built branches: ubuntu and macOS.

Figure 6: A data visualisation containing a sample of the names of the *contributors*. A total of **2278** *unique authors* have contributed with at least one source to our dataset.

This project is dealing with an impressive amount of data[7] which requires advanced techniques of data operation that need to scale. We likely *cannot* assume that our entire dataset will fit in the working memory. Only the **raw** metadata file containing the paths of all sources is over 50 MB. The challenges are not only in terms of managing the database of sources, rather than complex file-systems drawbacks caused by the high number of sources, assembly generated code and executable files. For example, NTFS performance severely degrades after 10000 files in a directory. Our approach was to try to create an additional levels in the directory hierarchy, with each subdirectory having under 10000 files. Yet, this isn't a NTFS-specific problem: in fact, it's commonly encountered on large UNIX systems. Another issue we encountered is that we operate with files that may be very small, typically less than the block sizes of the file-system, which led to additional required storage[8].

4.2 Synthetic inputs

While 140905 is the number of the sources that we managed to add to this dataset, this only reflects *part of what our project requires*. There are *over* 5000 problems for which we have obtained solution sources. In order to achieve *dynamic results* out of which we can create code embeddings, we need to create **synthetic test scenarios** for each problem individual project.

For this, we began a project called TheInputsCodeforces[9], which contains custom suite of

[7] As presented in Tables 4.1, 4.2, we are operating with over 300000 files stored on more than 16GB for each target operating system only in terms of source code.

[8] Almost all modern file-systems do allocate disk space for files in blocks. It is not possible to have a storage size less than one block (the default value of the size of our blocks were 4KB).

[9] TheInputsCodeforces is an open-source project: https://github.com/raresraf/TheInputsCodeforces.

inputs for various CodeForces problems. The inputs can be randomly generated from sample python scripts with respect to the requested input size. The simplest example of a generator, that takes into account the constrains for the problem *670A*[10] listed on Codeforces is the following:

```
import numpy as np

if __name__ == '__main__':
  test_cases = np.linspace(1, 1000000, num=50)
  for test_case in test_cases:
    with open(f"../{int(test_case)}.DAT", 'w') as f:
      f.write(f"{int(test_case)}")
```

Using the previous configuration for the generator, we can obtain a couple of synthetic variable inputs for corresponding input sizes:

```
"20409.DAT", // Sample synthetic file for an input size of 20409.
[...]
"448980.DAT","469388.DAT","489796.DAT","510204.DAT","530612.DAT",
[...]
"1000000.DAT" // Sample synthetic file for an input size of 1000000.
```

For this particular problem, the only input was a integer, less than 1000000. A larger generator is the one for problem **236A**, which should be a string of length at most 100:

```
import numpy as np
import random
import string

if __name__ == '__main__':
  test_cases = np.linspace(1, 100, num=50)
  for test_case in test_cases:
    with open(f"../{int(test_case)}.DAT", 'w') as f:
      s = ''.join(random.choice(string.ascii_lowercase) for _ in range(int(
    test_case)))
      f.write(f"{s}\n")
```

All these files will be used in building the associated embedding for the code snippet we analyze. While the presented generator seems to be trivial, there are cases in which generating inputs based on the size becomes a difficult task. In terms of competitive programming, problems with large graphs tends to generate bigger input files, and ensuring a high relevance of the critical code path in relation with the input size becomes an art.

[10]The full description of problem 670A can be consulted in the **Appendix C**.

Total number of problems	Total Number of synthetic inputs	Total number of analysis files
42	1803	1110636

Table 4.3: The size of the dataset containing the generated synthetic inputs and the corresponding number of resulted dynamic analysis files.

4.3 Analysis files

The output of the sources using as inputs the above generated synthetic files represents the profiling details of the programs analysed. These are created dynamically, by actually running the compiled programs. These are the main building blocks for establishing the embeddings as described in **Chapter 3**. Using the data generated by running (dynamic analysis) the sample code against various inputs, we can observe patterns in how the code behaves when the input size changes. More details about the procedure to follow.

The meaning of the metrics analyzed and captured in this files is configurable based on the tools that you use to crawl data. We have various default solutions, using two Linux tools: **/usr/bin/time**[11] and perf[12].

Sample output of **/usr/bin/time -v** under Linux shows monitoring metrics such as different time measurements, memory considerents and I/O operations. Full set of metrics available while running this utility is:

```
User time (seconds): 0.00
System time (seconds): 0.00
Percent of CPU this job got: 100%
Elapsed (wall clock) time (h:mm:ss or m:ss): 0:00.00
Average shared text size (kbytes): 0
Average unshared data size (kbytes): 0
Average stack size (kbytes): 0
Average total size (kbytes): 0
Maximum resident set size (kbytes): 3508
Average resident set size (kbytes): 0
Major (requiring I/O) page faults: 0
Minor (reclaiming a frame) page faults: 138
Voluntary context switches: 1
Involuntary context switches: 0
Swaps: 0
File system inputs: 0
```

[11] **/usr/bin/time** is a command that can measure "real,user,sys" elapsed seconds, but also other advanced features such as "maximum resident set size, page faults, swaps" and others.

[12] perf (originally Performance Counters for Linux) is a performance analyzing tool in Linux.

```
File system outputs: 0
Socket messages sent: 0
Socket messages received: 0
Signals delivered: 0
Page size (bytes): 4096
Exit status: 0
```

perf under Linux monitors a slightly different set of metrics, such as time measured on CPU, frequency of page-faults, cpu-migrations and context-switches, number of ran instructions and details about branch-misses. Sample output of **perf stat –all-user** under Linux:

```
    0,79 msec task-clock                #    0,663 CPUs utilized
       0      context-switches          #    0,000 K/sec
       0      cpu-migrations            #    0,000 K/sec
      52      page-faults               #    0,066 M/sec
 303.693      cycles                    #    0,384 GHz
 902.311      stalled-cycles-frontend   #  297,11% frontend cycles idle
 196.985      instructions              #    0,65  insn per cycle
                                        #    4,58  stalled cycles per insn
  45.454      branches                  #   57,460 M/sec
   4.041      branch-misses             #    8,89% of all branches

       0,001193778 seconds time elapsed

       0,001310000 seconds user
       0,000000000 seconds sys
```

Under **macOS** we used **fish**[13] to run Unix tools for gathering data. Sample output of **/usr/bin/time -lp** under Fish:

```
    544768  maximum resident set size
         0  average shared memory size
         0  average unshared data size
         0  average unshared stack size
       155  page reclaims
         0  page faults
         0  swaps
         0  block input operations
         0  block output operations
         0  messages sent
```

[13]Fish is a user-friendly command line shell for Linux, macOS, and the rest of the family.

```
0  messages received
0  signals received
1  voluntary context switches
3  involuntary context switches
```

While most of the entries are nil for trivial problems, the more advanced the problem solution or the size of the input is, the more meaningful these metrics become. Long running programs tend to show positive values for metrics such as page faults and context switches.

4.4 Dataset visualization

To perform a data analysis of our dataset in a 2D space, we used two main techniques for reducing our dataset dimensions to fit a two-dimension representation: Self-organizing Maps and Linear discriminant analysis.

4.4.1 Self-organizing Maps: Intro

We have prepared two datasets to perform the data visualisation on:

- $math\_dataset.csv$ containing 5949 code-embeddings, out of which 4937 for problems that are not related to math and 1012 for problems that are classified as math-related.
- $small\_math\_dataset.csv$, a subset of $math\_dataset.csv$, containing 200 code-embeddings, out of which 100 for problems that are not related to math and 100 for problems that are classified as math-related.

A sample dataset entry, containing 2 code-embeddings, 1 for a problem that is not related to math and 1 for a problem that is not classified as math-related:

```
[CSV columns names]

branch-misses_FEATURE_CONFIG,branch-misses_FEATURE_TYPE,
branch-misses_INTERCEPT,branch-misses_R-VAL,
branches_FEATURE_CONFIG,branches_FEATURE_TYPE,
branches_INTERCEPT,branches_R-VAL,
context-switches_FEATURE_CONFIG,context-switches_FEATURE_TYPE,
context-switches_INTERCEPT,context-switches_R-VAL,
cpu-migrations_FEATURE_CONFIG,cpu-migrations_FEATURE_TYPE,
cpu-migrations_INTERCEPT,cpu-migrations_R-VAL,
cycles_FEATURE_CONFIG,cycles_FEATURE_TYPE,
```

```
cycles_INTERCEPT,cycles_R-VAL,
instructions_FEATURE_CONFIG,instructions_FEATURE_TYPE,
instructions_INTERCEPT,instructions_R-VAL,
page-faults_FEATURE_CONFIG,page-faults_FEATURE_TYPE,
page-faults_INTERCEPT,page-faults_R-VAL,
stalled-cycles-frontend_FEATURE_CONFIG,stalled-cycles-frontend_FEATURE_TYPE,
stalled-cycles-frontend_INTERCEPT,stalled-cycles-frontend_R-VAL,
task-clock_FEATURE_CONFIG,task-clock_FEATURE_TYPE,
task-clock_INTERCEPT,task-clock_R-VAL,
label

[entry 1]
0,LOG_POLYNOMIAL,12176.278639860398,5.274471559958294,
1,POLYNOMIAL,354489.8418811041,31.023435300379777,
0,LOG_POLYNOMIAL,0.0,0.0,
0,LOG_POLYNOMIAL,0.0,0.0,
3,LOG_POLYNOMIAL,1657272.38324269,1.4722602961244892e-05,
1,POLYNOMIAL,2196944.89849295,137.26687233920643,
0.9,FRACTIONAL_POWER,119.36451460061674,0.7558436224332522,
5,LOG_POLYNOMIAL,2277557.4855760685,1.994809269682738e-11,
3,POLYNOMIAL,2.470760207032896,-2.172036539848529e-10,
[label 1]
0

[...]

[entry 200]
1,POLYNOMIAL,12370.712368796743,0.9930555401849003,
1,POLYNOMIAL,355267.67041624436,818.7959073312177,
0,LOG_POLYNOMIAL,0.0,0.0,
0,LOG_POLYNOMIAL,0.0,0.0,
1,POLYNOMIAL,1673084.5351177657,1735.1433240315066,
1,POLYNOMIAL,2201813.5172021347,3798.310753165602,
0,LOG_POLYNOMIAL,118.69266672203334,0.06013108223346909,
1,POLYNOMIAL,2315580.396807521,1911.1634798457628,
1,LOG_POLYNOMIAL,2.3873598897146207,0.00025835398225644443,
[label 200]
1
```

Self-organizing maps (**SOM**) are a powerful visualization mechanism for investigating the dataset. Learning in the self-organizing map aims to **make various regions of the network respond to certain input patterns in a similar way**. This is largely inspired from of the

way visual, auditory, and other sensory information is interpreted in different segments of the human brain's cerebral cortex. This statement has been sustained in research by [22] and [10].

Quantization error and **topographical error** are main measurements to assess the quality of SOM. **Quantization error** is the average difference of the input samples compared to its corresponding winning neurons (BMU, best matching unit). It assesses the accuracy of the represented data, therefore, it is better when the value is smaller [14]. **Topographical error** assesses the topology preservation. It indicates the number of the data samples having the first best matching unit (BMU1) and the second best matching unit (BMU2) being not adjacent. Therefore, the smaller value is better [13].

Using these two metrics, we will evaluate the quality of the resulted feature map after the learning process for the self-organizing map for the different test scenarios that we have prepared, for the reduced/entire dataset.

4.4.2 Self-organizing Maps: Reduced dataset

Optimal Neighborhood function

We have performed a 100k-iteration long training for various Neighborhood function, using the entire feature set available for this reduced dataset:

- **gaussian**
- mexican-hat
- bubble
- triangle

The most relevant results were obtained when using the **gaussian** function, as presented in the below graphs. This has outputted the lowest quantization error and the best spread of dataset entries across multiple neurons.

Optimal Activation Distance Function

We have performed a 100k-iteration long training for various Activation Distance, using the entire feature set available for this reduced dataset:

- **euclidean**
- **cosine**
- manhattan
- chebyshev

The most relevant results were obtained when using the **euclidean** and **cosine** function, as presented in the below graphs. These two activation functions have outputted the lowest quantization error and the best spread of dataset entries across multiple neurons.

Feature selection

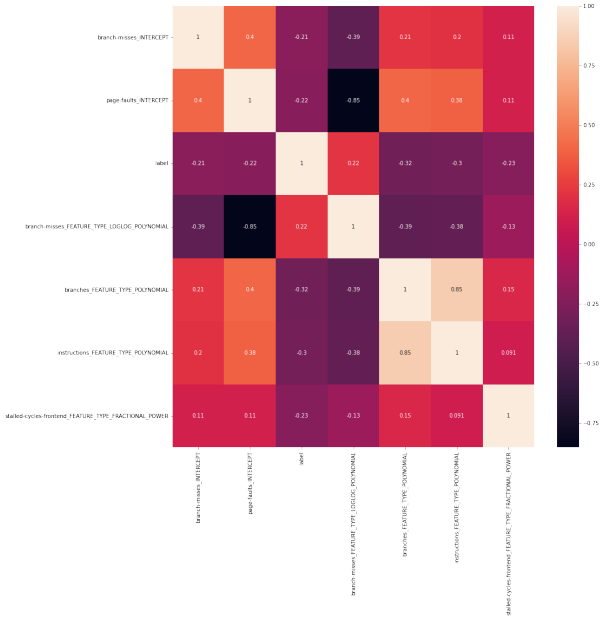
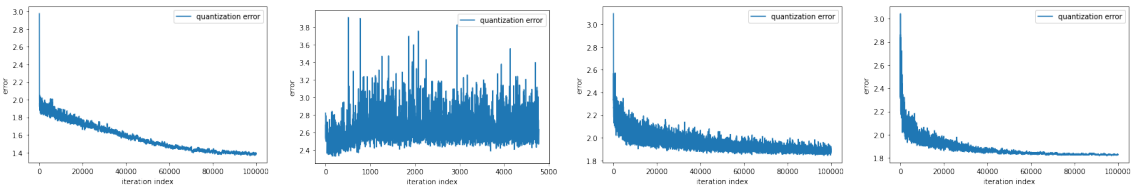
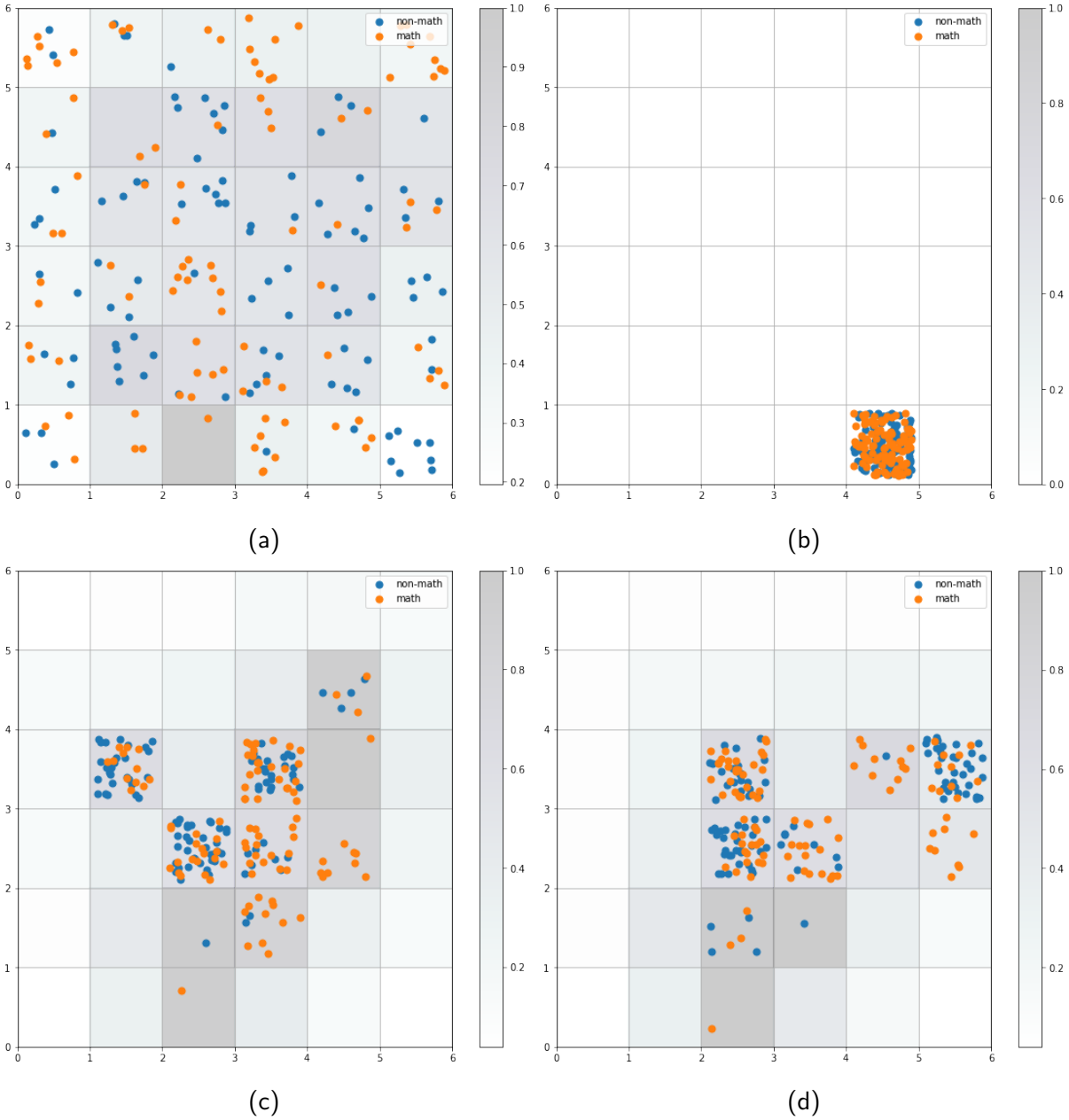


Figure 11: Heatmap corresponding to the top features that have a Pearson correlation (in absolute value) with the targeted label of at least **.2**.



(a) (b) (c) (d)

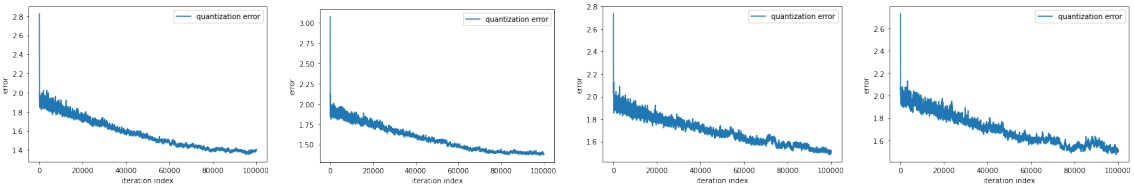
Figure 7: The Quantization error obtained when using the Neighborhood function: (a) **gaussian** (b) **mexican-hat** (c) **bubble** (d) **triangle**, during a 100k iteration training of a 6x6 SOM.



(a) (b)

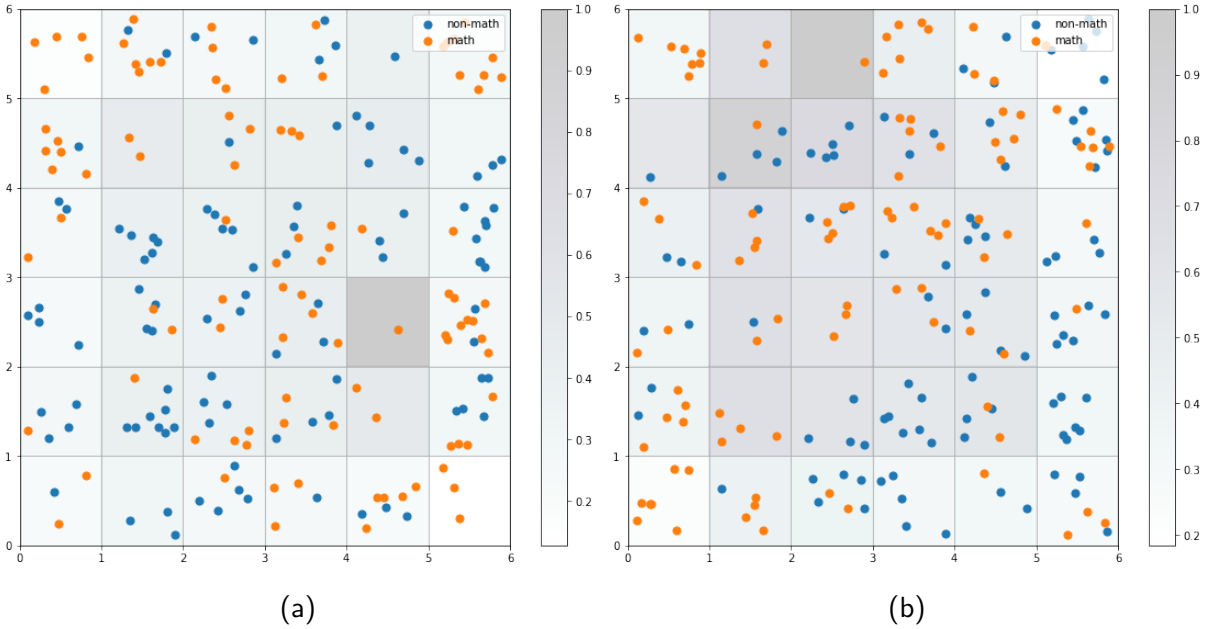
(c) (d)

Figure 8: The distribution across the self-organizing map of the dataset obtained when using the Neighborhood function: (a) **gaussian** (b) **mexican-hat** (c) **bubble** (d) **triangle**, during a 100k iteration training of a 6x6 SOM.

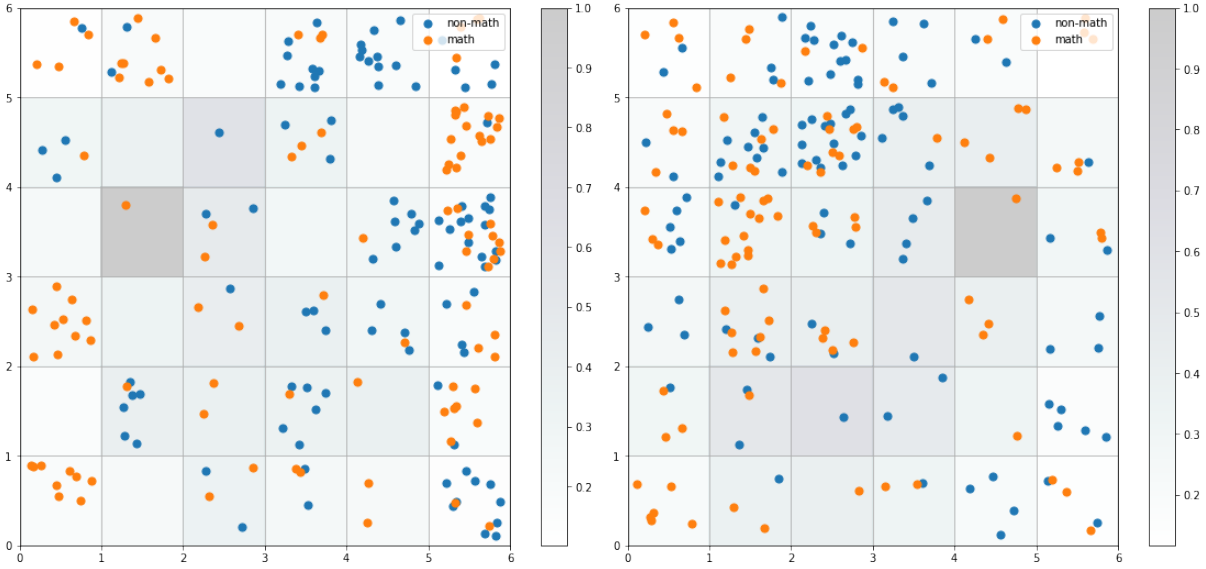


(a) (b) (c) (d)

Figure 9: The Quantization error obtained when using the Activation function: (a) **euclidean** (b) **cosine** (c) **manhattan** (d) **chebyshev**, during a 100k iteration training of a 6x6 SOM.



(a) (b)



(c) (d)

Figure 10: The distribution across the self-organizing map of the dataset obtained when using the Activation function: (a) **euclidean** (b) **cosine** (c) **manhattan** (d) **chebyshev**, during a 100k iteration training of a 6x6 SOM.

We ran 1M training-iteration using the Gaussian Neighbourhood function on the small dataset, containing:

- All features provided in the dataset.
- Selected features provided in the dataset, chosen based on correlation factors. To begin with, we investigated the correlations between the features and the feature of the targeted algorithm in the dataset. We have used the pandas implementation of the Pearson correlation.
  The selected features were:
  - branch-misses_INTERCEPT,
  - page-faults_INTERCEPT,
  - branch-misses_FEATURE_TYPE_LOGLOG_POLYNOMIAL,
  - branches_FEATURE_TYPE_POLYNOMIAL,
  - instructions_FEATURE_TYPE_POLYNOMIAL,
  - stalled-cycles-frontend_FEATURE_TYPE_FRACTIONAL_POWER.

We have summarized the results for both scenarios in the following subsubsections.

Using all dataset features

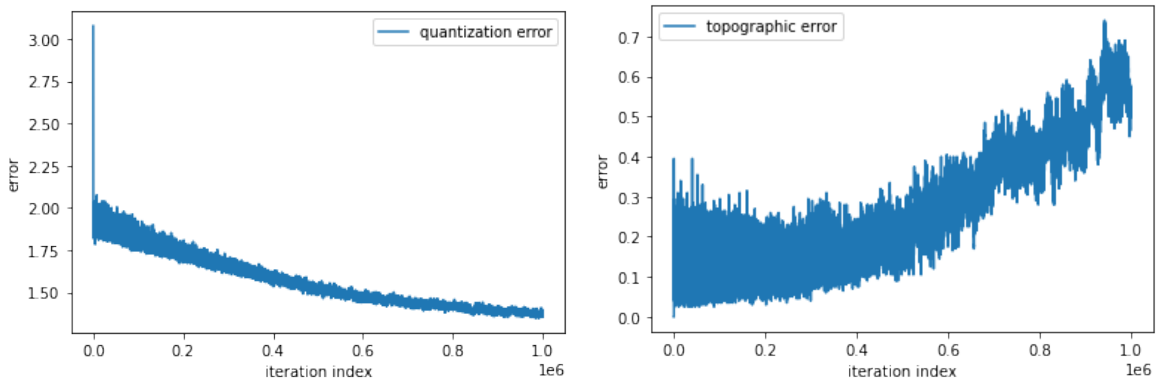


Figure 12: Quantization and Topographic error of the SOM when performing on the small dataset with all features.

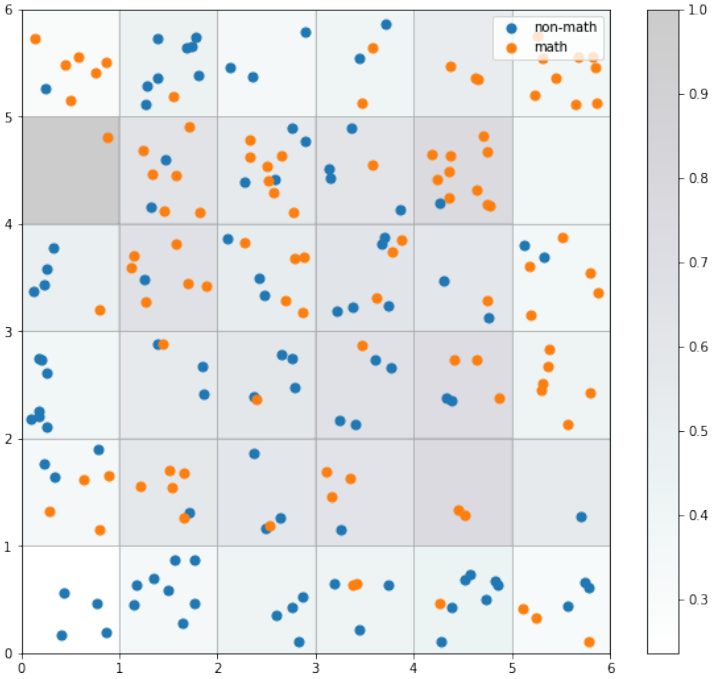


Figure 13: [Small dataset - All features] To visualize the result of the training we can plot the distance map (U-Matrix) using a pseudocolor where the neurons of the maps are displayed as an array of cells and the color represents the (weights) distance from the neighbour neurons. To have an overview of how the samples are distributed across the map a scatter chart can be used where each dot represents the coordinates of the winning neuron.

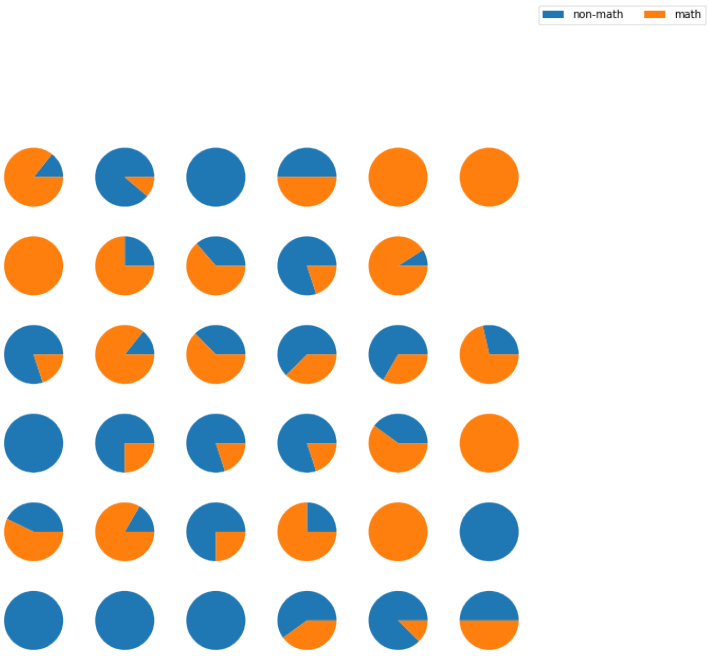


Figure 14: [Small dataset - All features] We can visualize the proportion of samples per class falling in a specific neuron using this pie chart per neuron.

Using some dataset features

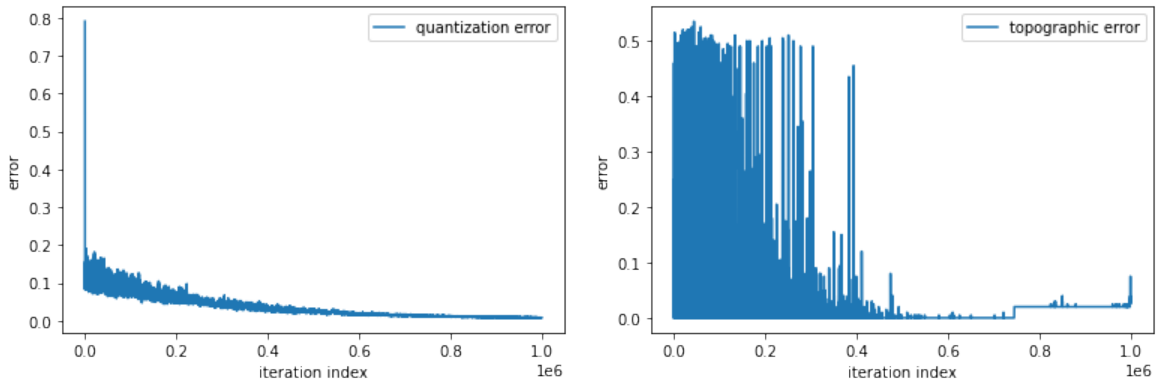


Figure 15: Quantization and Topographic error of the SOM when performing on the small dataset with selected features.

4.4.3 Self-organizing Maps: Large dataset

For this experiment, using the large, containing over 5000+ entries, we have used only a subset of the dataset features with high correlation, gaussian neighborhood function and cosine function as an activation in the SOM model. We trained a 216x216 SOM network for 10M steps.

The results are available open-source[14].

[14] https://github.com/raresraf/AlgoRAF/blob/master/prototype/SelfOrganizingMap/allDataset-SOM-SomeFeatures.ipynb

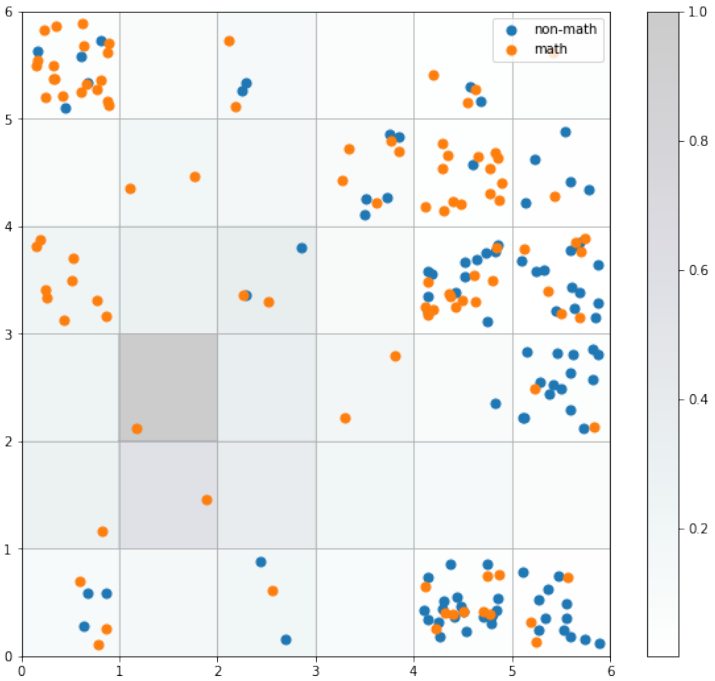


Figure 16: [Small dataset - Selected features] To visualize the result of the training we can plot the distance map using a pseudocolor where the neurons of the maps are displayed as an array of cells and the color represents the (weights) distance from the neighbour neurons. To have an overview of how the samples are distributed across the map a scatter chart can be used where each dot represents the coordinates of the winning neuron.

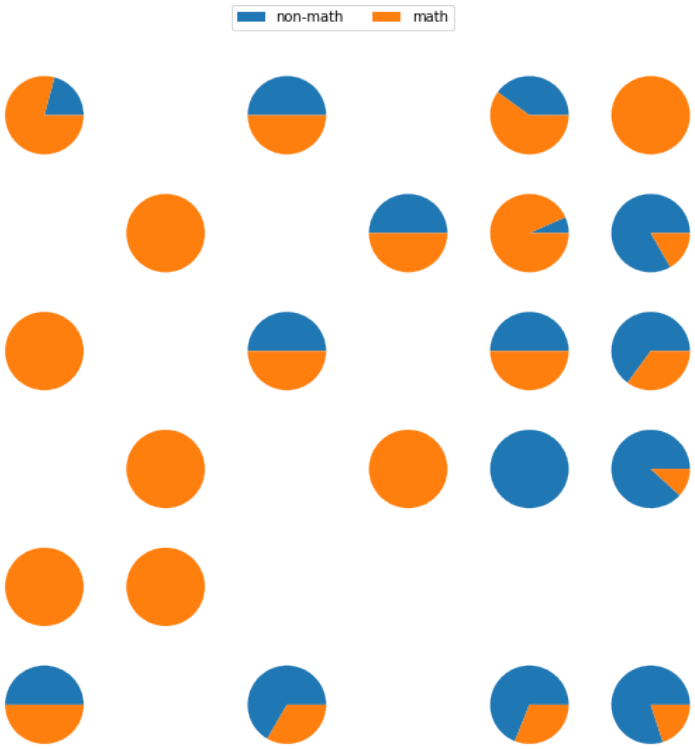


Figure 17: [Small dataset - Selected features] We can visualize the proportion of samples per class falling in a specific neuron using this pie chart per neuron.

4.4.4 Linear Discriminant Analysis: 2D projection

In the case of Linear Discriminant Analysis, the model tries to identify attributes that account for the most variance between the analyzed classes.

```
import matplotlib.pyplot as plt
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis

# Define the LinearDiscriminantAnalysis model.
lda = LinearDiscriminantAnalysis(n_components=2)

# Obtain the 2D dimensions.
X_r2 = lda.fit(X, y).transform(X)
```

Listing 4.1: Snippet for obtaining the 2D LDA projection of the dataset.

Linear discriminant analysis is similar to principal component analysis, as both methods try to look at linear combinations of variables that best explain the data. LDA makes a clear effort to model the differences between data classes, while PCA ignores class differences.

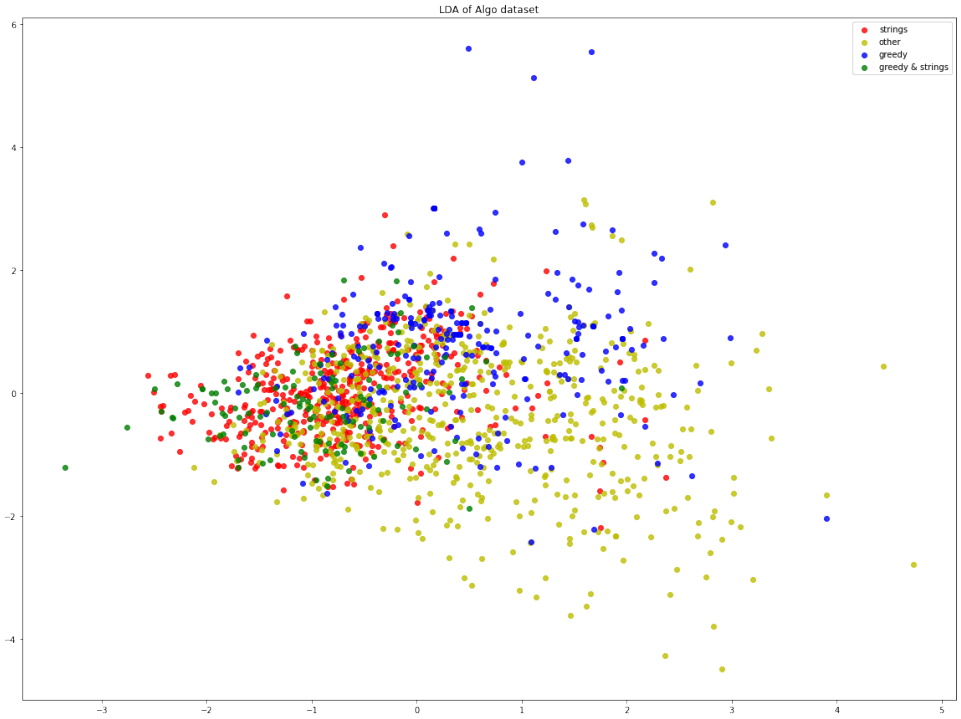


Figure 18: 2D LDA projection of the dataset, highlighting the split between greedy and string problems, using LDA technique

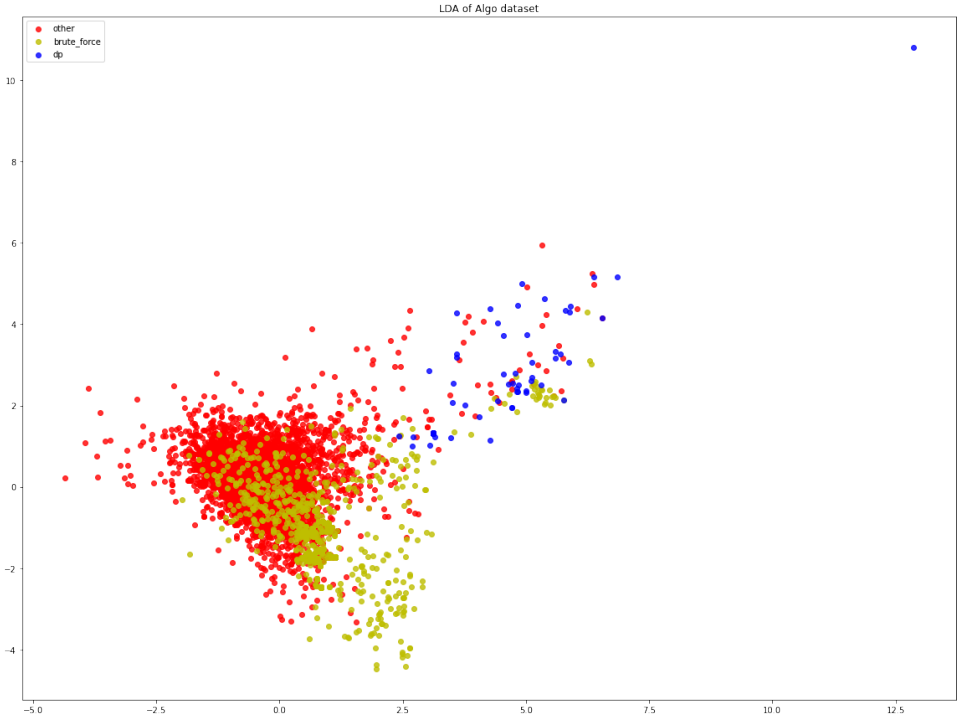


Figure 19: 2D LDA projection of the dataset, highlighting the classification difference between brute force and dynamic programming problems, using the LDA technique

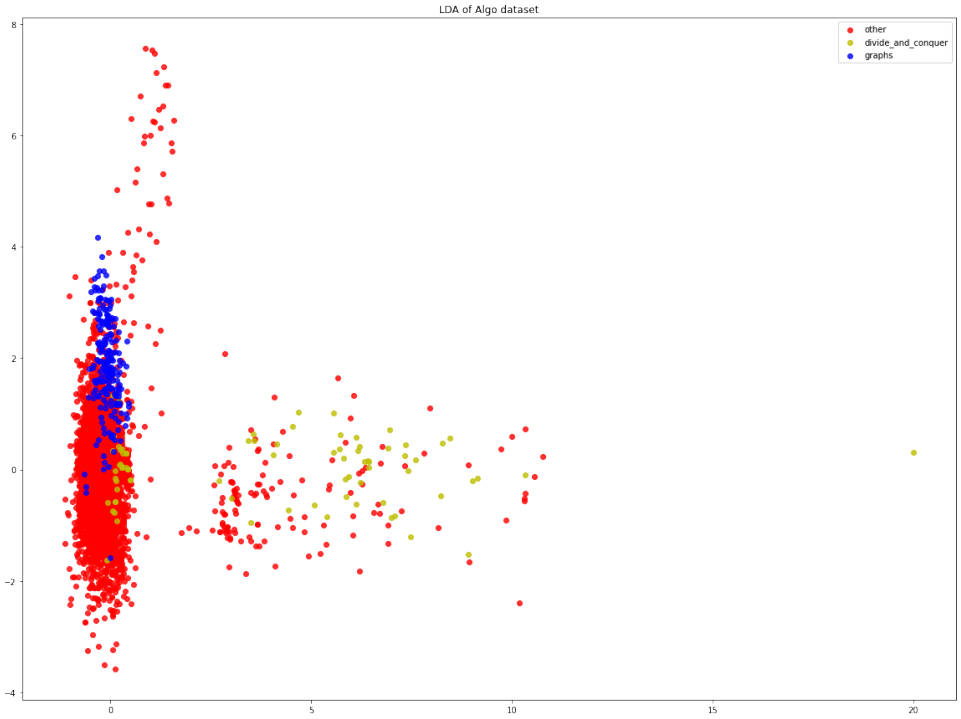


Figure 20: 2D LDA projection of the dataset, highlighting the split between divide and conquer problems against graphs

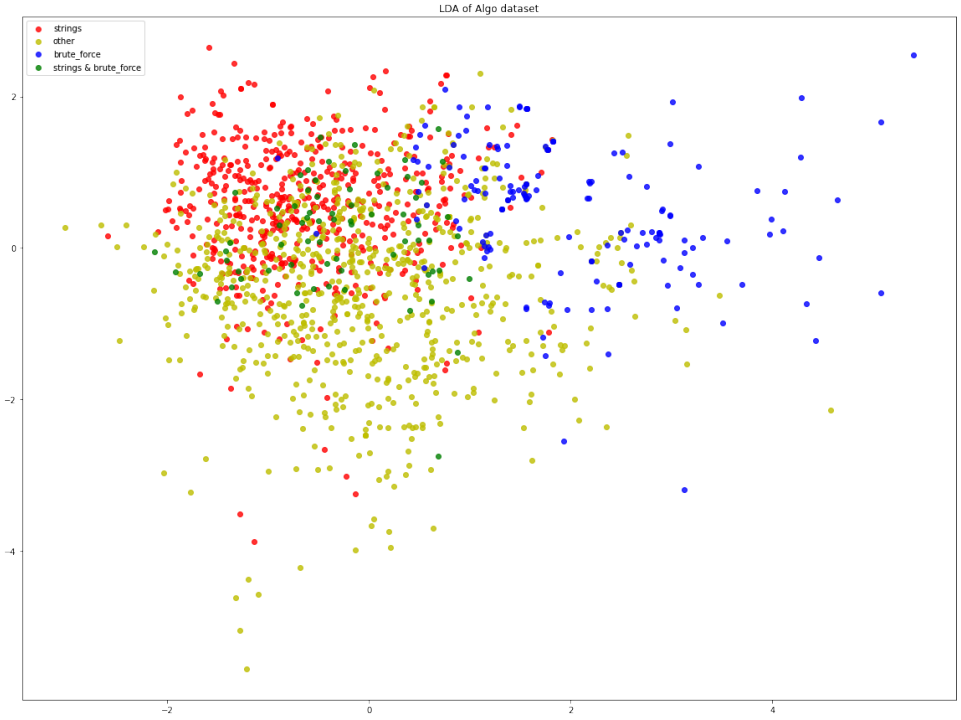


Figure 21: 2D LDA projection of the dataset, highlighting brute force vs strings splits.

Chapter 5

System

The *system* we proposed to analyse the solutions to the problems contained in the dataset presented above is complex. It can be seen as two distinct entities: **the data acquisition subsystem**, that consists of two main parts, the **Control Plane** and the **Data Pipeline Plane** and the **embeddings subsystem**.

Both subsystems will be in-depth analysed in their corresponding sections.

5.1 The Data Acquisition System

The **Control Plane** exposes public APIs that can be used by the administrators or users of our system. It also contains the internals that allows our pipeline to operate. It is responsible of the control flow of the pipelines, as well as storing the metadata of the datasets and the configurations. This is the system that governs what happens on the Data Pipeline Plane. Moreover, it can be used to generate various operational reports.

The **Data Pipeline Plane** is a core part of our system that is responsible of obtaining the raw metrics used by our models. It aggregates the problem set and fetches the solutions to it from **TheCrawlCodeforces**, a repository that is read-only for raw source files problems (with .CPP extension), but writes are permitted for assembly and binary files (each operating machinery used in this will write to its own branch). The raw source files are computed concurrently. An executable corresponding to a specific problem is then ran against corresponding synthetic input files available on-demand on **TheInputsCodeforces** repository. During the execution, predefined profilers or monitors audits the usage of various system resources as well as live time monitoring of the operations. This contains for each problem a generated problem-specific set of inputs, which are **sequentially** ran on the machine against all given solution executable. The results are stored both raw and aggregated on **TheOutputsCodeforces** repository, which ends our pipeline.

An interesting part of our system is the crawling part, which supports continuous updates for

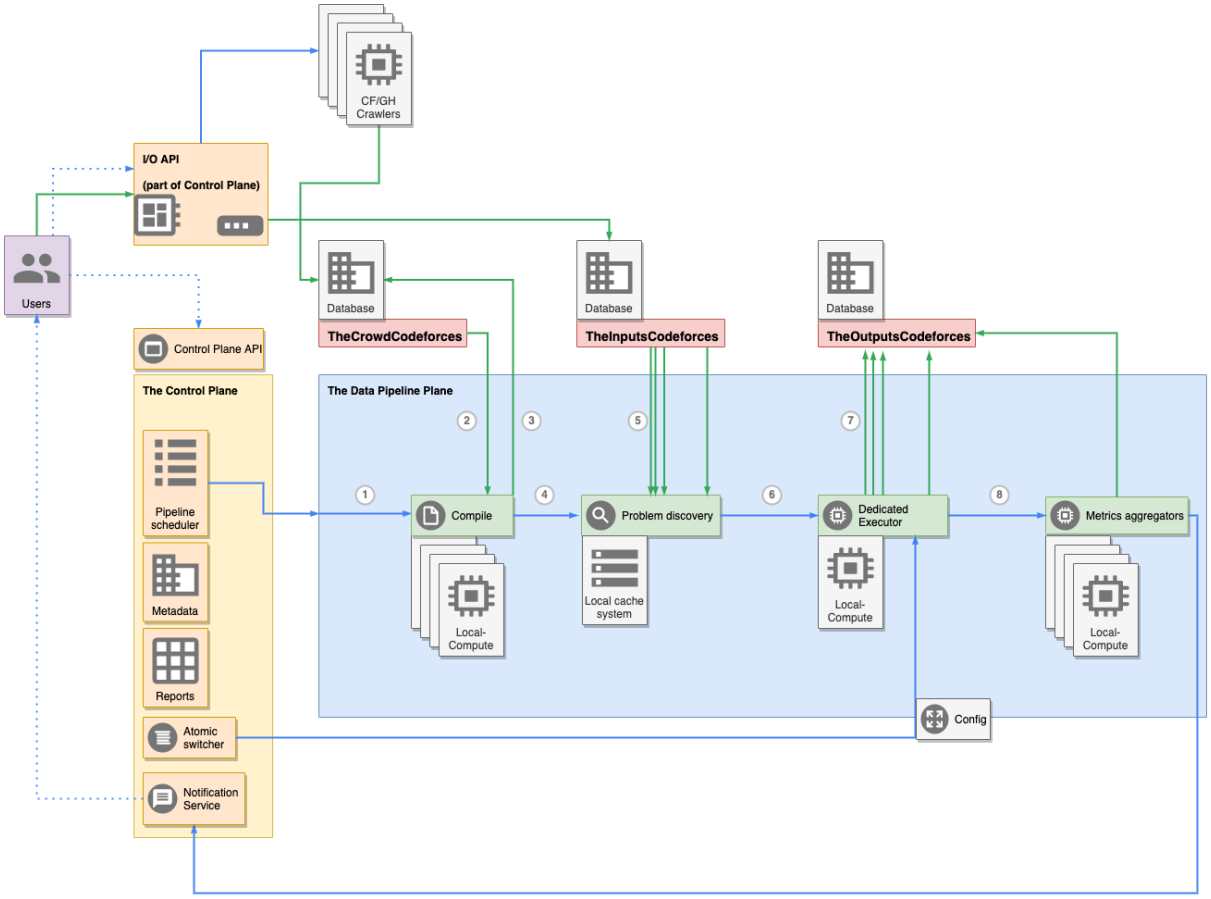


Figure 22: An overview of our data acquisition system. The path of running a pipeline is illustrated by the labels. The Control Plane initiates a data pipeline in **1**. Step **2** fetches the solutions to the specific problem from TheCrawlCodeforces, compiles and store the results back in **3**. Step **4** is passing the compiled executable, to the discovery service, which fetches synthetic inputs from TheInputsCodeforces in **5**. The execution in started on a dedicated executors in **6**, and write the profiling data in **7** to TheOutputsCodeforces. In the end of the pipeline, metrics are being aggregated in **8** and the pipeline is marked as succeeded.

the **TheCrawlCodeforces** repository. It can be ran on-demand to fetch more source files. **TheInputsCodeforces** supports writing custom generators at any given time. This can be done independently to the main data pipeline, while extending this dataset will increase the options of the discovery service, thus extending the main dataset.

The metrics reported and stored in **TheOutputsCodeforces** repository are formatted using JSON, containing metadata regarding the ownership of the file, the size of the input and the actual analysed metrics. Merging multiple JSON provide the data needed in order to tailor custom complexity functions. This will be the core of the custom embeddings that we will create for an arbitrary code snippet. Based on these code embeddings, we created the custom models for automatic code labelling using machine learning techniques.

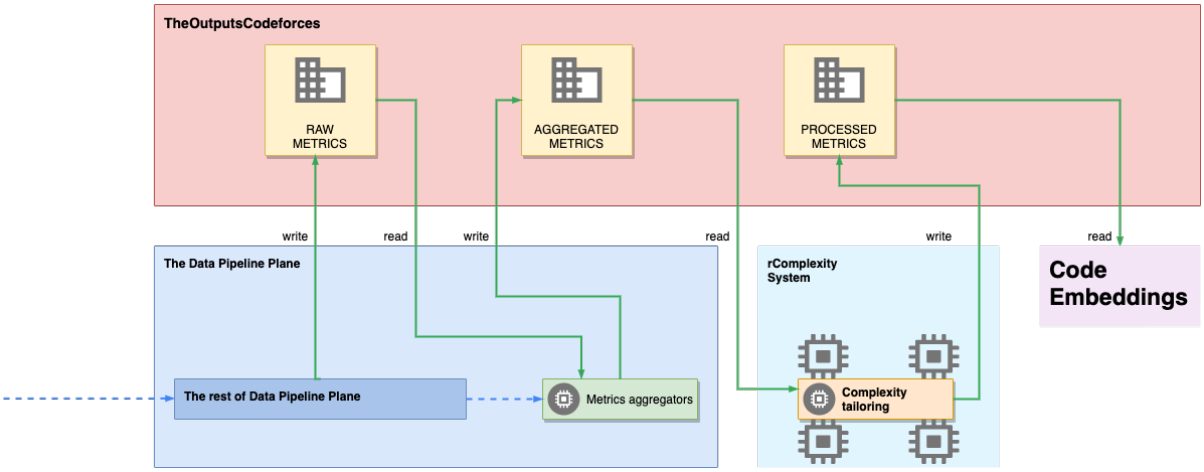


Figure 23: An overview of our data embedding system and how it interacts with other parts of the systems.

5.2 The Embeddings System

The embeddings system is responsible of building code embeddings from the available data provided by the Data Acquisition System. It is responsible of computing at scale dynamic code-embeddings based on r-Complexity 3 wrt. various metrics analyzed against the input dimensions.

The profiling data in stored to TheOutputsCodeforces as part of the Data Pipeline Plane and metrics are being aggregated. Using the aggregated metrics, we can compute estimation for suitable r-Complexity for each metric, using the general tailoring functions described in **Section 2.3.2**.

For our usage of features inside the embeddings, we have only used the results of profiling[1] performed under Linux:

```
perf stat -all-user
```

As stated in **Chapter 3**, these embeddings can be built on top of any metrics collected. Using the above usage of `perf`, we obtained and used the following metrics:

- branch-misses
- branches
- context-switches
- cpu-migrations
- cycles
- instructions
- page-faults
- stalled-cycles-frontend
- task-clock

Therefore, based on these metrics, our code can be processed as a 36-long dynamic-code

[1] Using perf, a performance analyzing tool for Linux

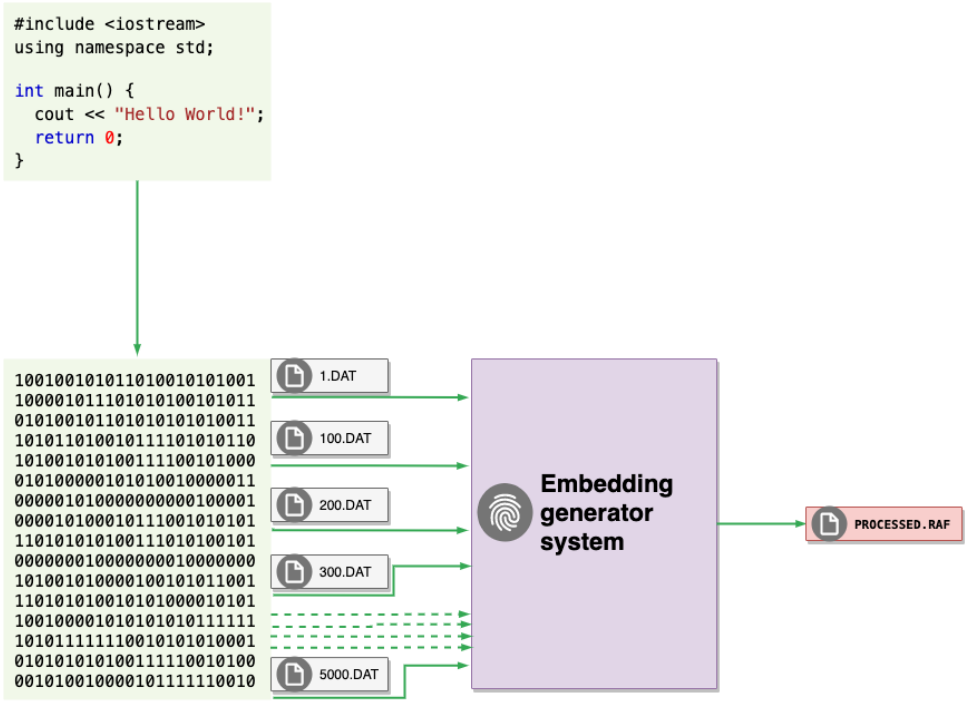


Figure 24: The simplified process of generating a code-embedding based on r-Complexity using this system

embedding[2], built using approximations of r-Theta complexity, containing:

- branch-misses_FEATURE_CONFIG
- branch-misses_FEATURE_TYPE
- branch-misses_INTERCEPT
- branch-misses_R-VAL
- branches_FEATURE_CONFIG
- branches_FEATURE_TYPE
- branches_INTERCEPT
- branches_R-VAL
- context-switches_FEATURE_CONFIG
- context-switches_FEATURE_TYPE
- context-switches_INTERCEPT
- context-switches_R-VAL
- cpu-migrations_FEATURE_CONFIG
- cpu-migrations_FEATURE_TYPE
- cpu-migrations_INTERCEPT
- cpu-migrations_R-VAL
- cycles_FEATURE_CONFIG
- cycles_FEATURE_TYPE
- cycles_INTERCEPT
- cycles_R-VAL
- instructions_FEATURE_CONFIG
- instructions_FEATURE_TYPE
- instructions_INTERCEPT
- instructions_R-VAL
- page-faults_FEATURE_CONFIG
- page-faults_FEATURE_TYPE
- page-faults_INTERCEPT
- page-faults_R-VAL
- stalled-cycles-frontend_FEATURE_CONFIG
- stalled-cycles-frontend_FEATURE_TYPE
- stalled-cycles-frontend_INTERCEPT
- stalled-cycles-frontend_R-VAL
- task-clock_FEATURE_CONFIG
- task-clock_FEATURE_TYPE
- task-clock_INTERCEPT
- task-clock_R-VAL

[2]We also provide a solution for generating a different set of embeddings, based on the time utility under Linux. Yet, these embeddings provide better results for the dataset that we have analyzed.

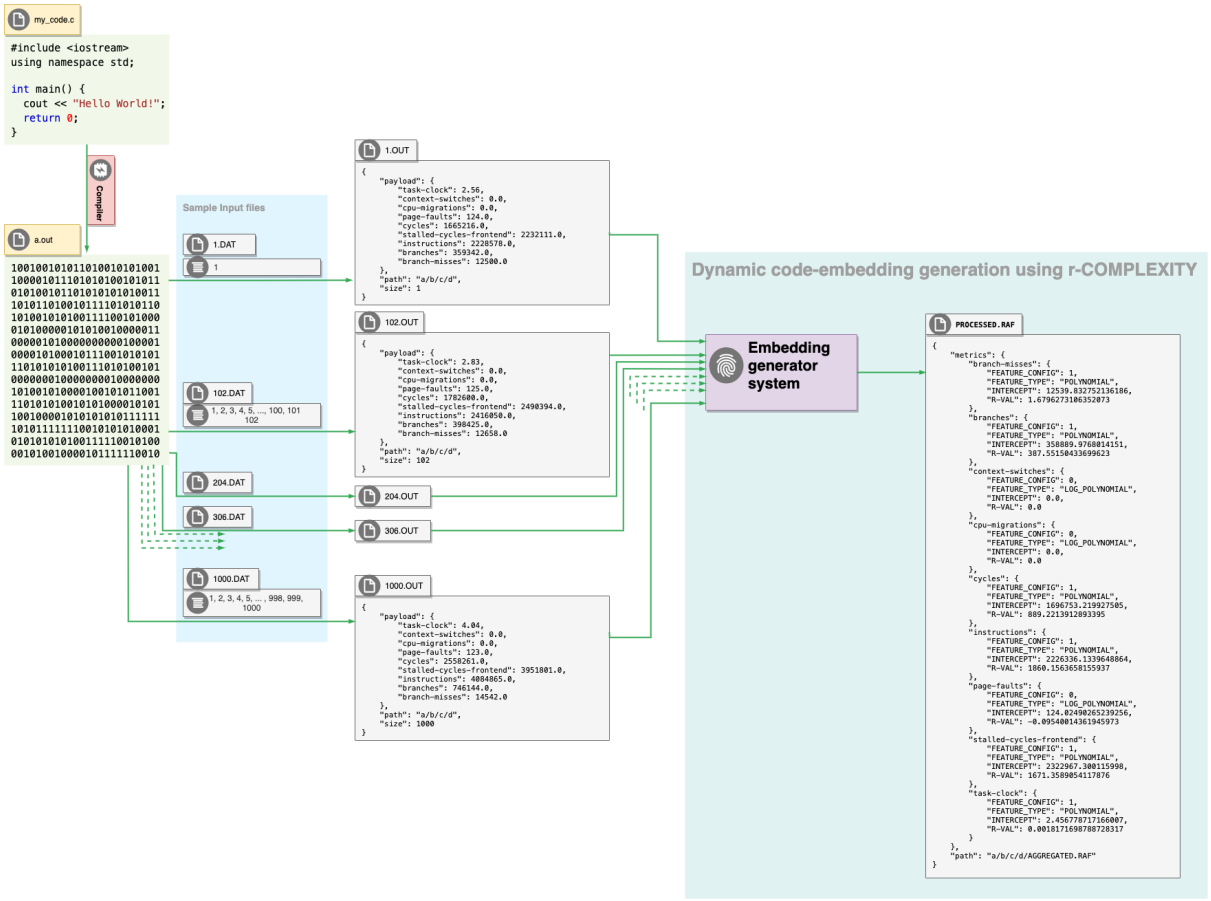


Figure 25: An in-depth view on the process of generating a code-embedding for a given algorithm, based on r-Complexity code-embeddings, using this system.

Sample embedding generated by the system can be consulted in **Appendix D**. The possible values for each feature is documented in **Chapter 3**, where presented the solution for discretizing the search space.

Chapter 6

Using the code embeddings for classification

All the previous chapters have focused on techniques of capturing the semantics of computer programs into arrays of numerical values and acquiring a relevant set of solutions to annotated competitive programming problems.

From now on, after the conversion between source code and code embeddings, the task of doing automatic labeling for competitive programming challenges is nothing but a **classification algorithm**, a *supervised learning* method that uses the set of acquired data, to determine the class of the new observations.

The exhaustive list[1] of available labels in the dataset are: flows, dp & **greedy**, dfs and similar, time, set, all pairs, **graphs**, dfs, bitmasks, array, bellmanford, stack, trees, number system, grid, suffix tree, game theory, flow, graph matchings, 2-sat, knapsack, disjoint sets, schedules, mst, priority queue, priorityqueue, fractions, graph, recurrence, shortest-path, sorting, matrices, big integer, bst, optimization, greedy, base, kuhn, counting, combinatorics, **divide and conquer**, geometry, games, tree, palindromes, data structures, bit manipulation, bfs, properties, cut, real life, **brute force**, log/exp/pow, matrix pow, fft, **binary search**, modulo, queue, cycle, scc, dsu, eulerian, negative weights, euclid, interactive, string suffix structures, **implementation**, lis, dag, 2d array, tree ds, **shortest paths**, backtracking, two pointers, expression parsing, range sum, hashing, dijkstra, **strings**, topological sort, gcd, *special problem, lca, probabilities, primes, ternary search, map, josephus problem, factorial, **math**, **dp**, cg, topsort, reachability, bipartite, flood fill, **sortings**, tsp, chinese remainder theorem, game, shortest path, meet-in-the-middle, sssp, number theory, maths, constructive algorithms, polynomial.

[1] In our study, we have only selected a subset of these labels, for which we managed to get a number of at least 75 support examples.

6.1 Decision Tree Classifier. Random Forest Classifier.

We have built a **decision tree classifier** and a **random forest classifier**.

A **decision tree** is a classifier, that has the objective to learn some basic decision rules, based on the data attributes, in order to build a model that predicts the value of a new observation variable. Each tree in a **random forest ensemble** is constructed using a sample selected with replacement from the training set.

For this example, we refer only to the task of classifying problems as math/non-math, against the existing dataset. An example of a math problem can be checked in **Appendix E**.

We have analyzed the performance of the models based on multiple metrics: **accuracy**, **precision**, **recall** and **f1-score**. There are two ways in which we have performed the split between training and testing. In first attempt, we have splitted it at solution level, with the only constraint being that no solution belongs both in the training and testing dataset. In second methodology, we have performed the split at problem level, by adding the constraint that if the solution belongs in the testing dataset, then no other solution from the same problem should belong in the training dataset. Obviously, less relevant information is being acquired by the machine learning classifier in the latter, so a performance drop was expected in this scenario. Nonetheless, it is interesting to test this as well, as such procedure covers a significantly more difficult problem to solve.

6.1.1 Testing at solution level

	precision	**recall**	**f1-score**	**support**
non-math	**0.98**	0.98	**0.98**	1663
math	0.89	**0.87**	0.88	301

Table 6.1: The obtained metrics for the **decision tree classifier** against the testing dataset, splitted at solution level. For training, we have used two thirds of the available data in the dataset and thus leaving the last third of the dataset for testing purposes.

	precision	**recall**	**f1-score**	**support**
non-math	0.97	**0.99**	**0.98**	1663
math	**0.96**	0.83	**0.89**	301

Table 6.2: The obtained metrics for the **random forest classifier** against the testing dataset, splitted at solution level. For training, we have used two thirds of the available data in the dataset and thus leaving the last third of the dataset for testing purposes.

The accuracy was similar: **96%** for the *decision tree classifier* and **97%** for the *random forest classifier*. Overall, only the recall was smaller for the random forest classifier, while the f1-score was slightly higher for this classifier.

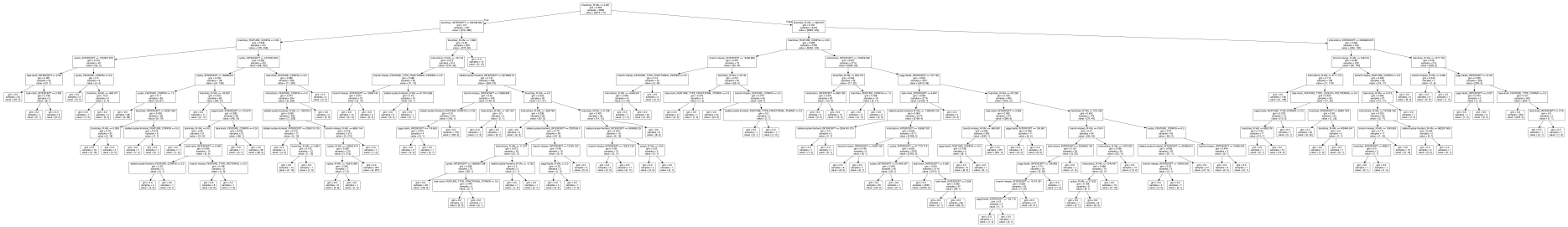


Figure 26: A sample decision tree classifier, that achieves over 95+% accuracy on the presented dataset. It takes into account multiple metrics, such as the variation of branches, the number of executed instructions, the number of page-faults, the total duration, etc. Full scale image available: `https://github.com/raresraf/AlgoRAF/blob/master/prototype/dtree/dtree_render.png`

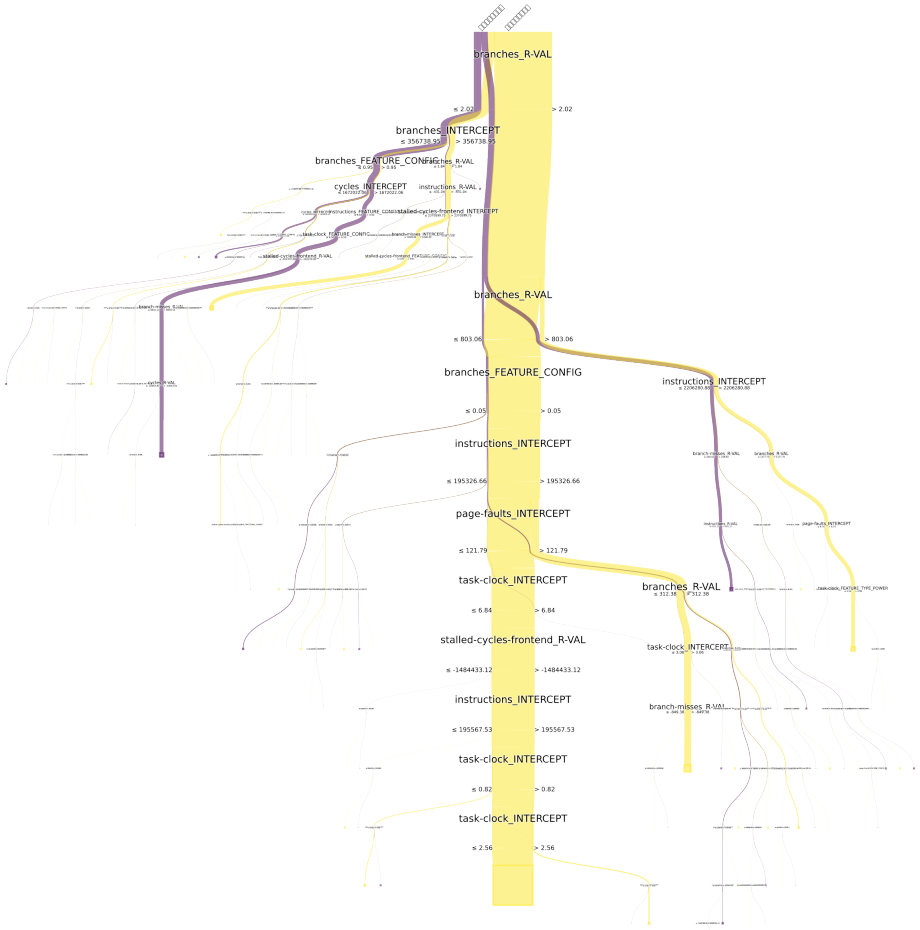


Figure 27: The decision tree classifier visualized using pybaobab [21]. Such a classifier has achieved over 96% accuracy on the task of labelling algorithmic challenges with the math/non-math label against our testing dataset, while trained on dataset we prepared, with open-source Codeforces submissions.

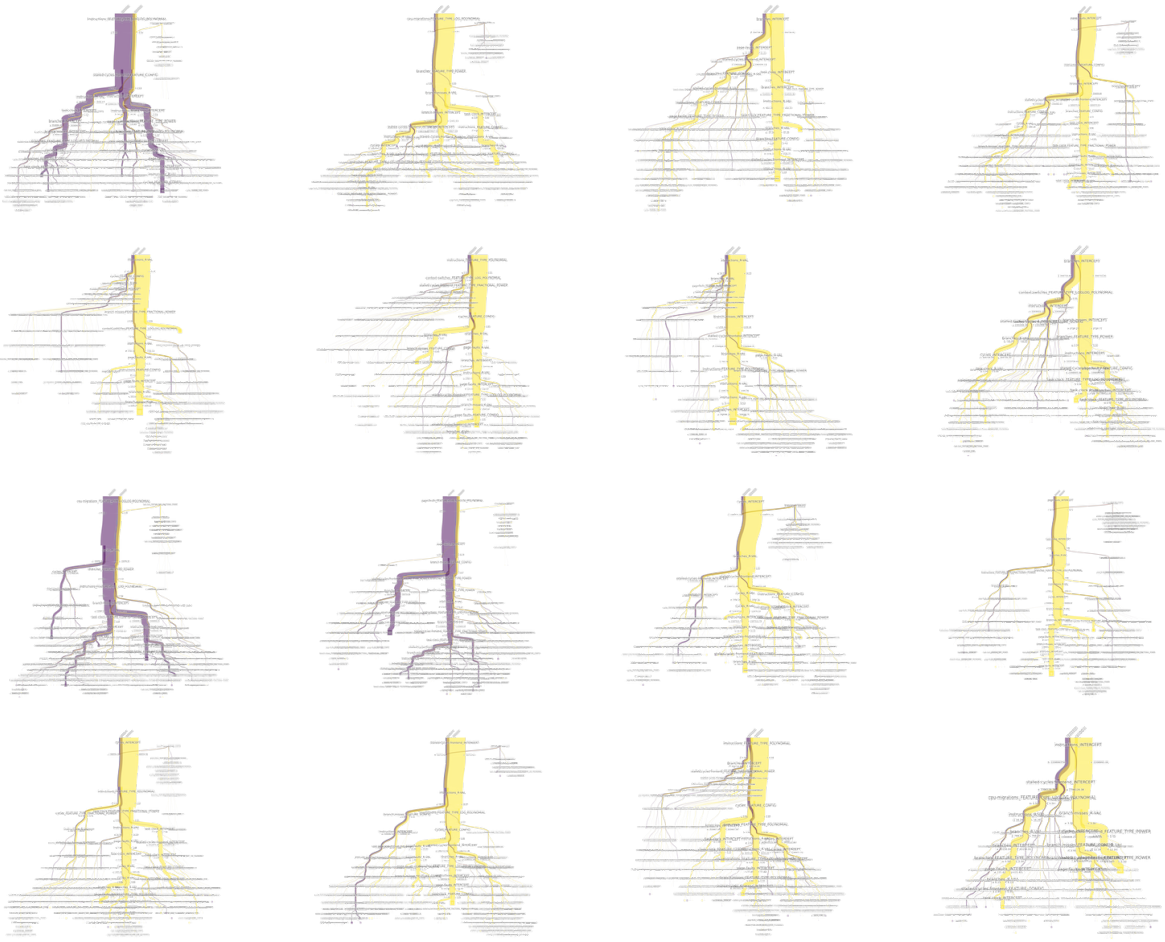


Figure 28: The random forest classifiers visualized using pybaobab [21]. Such a classifier has achieved over 97% accuracy on the task of labelling algorithmic challenges with the math/non-math label against our testing dataset, while trained on the dataset we prepared, with open-source Codeforces submissions.

Also, we have trained, using the same testing scenario, a (custom) deep neural network using this dataset, but the preliminary performance of such a classifier was unsatisfactory, as described in the dedicated sections, for the other classifiers. Tree-based approaches still yielded the best results.

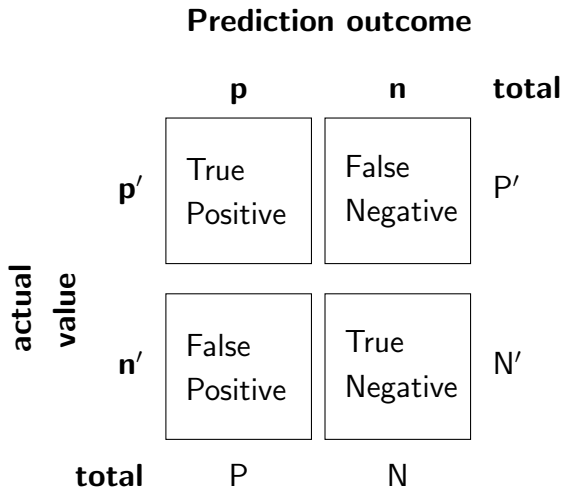
6.1.2 Testing at problem level

This is a much more difficult task, as no solutions from the testing dataset have been previously used in the training dataset. We have benchmark-ed our classifiers against multiple splits of the range of problems into training and testing.

Here, the results ranged considerably, based on the split of the problems. The split has been made randomly, using $2/3$ data-points for training purposes the rest of $1/3$ for testing. While splitting based on several seeds, some have returned particularly good results, while others returned less-satisfactory ones.

With a *93 per-cent accuracy* and a *f1-score of 0.82*, a decision tree classifier was able to correctly output the classification between math/non-math problem.

However, other splits have outputted a lower values of the f1-score, ranging, on average, between **0.35**-**0.8**. In **Table 6.2**, we present some results (with the seeds used in splitting the dataset) obtained using this testing scenario scenario.



Seed	F1-Score	Report	
		True Positive	False Negative
		False Positive	True Negative
19	0.54	1356	48
		180	134
85	0.62	991	214
		83	244
103	**0.82**	1351	59
		53	257
291	0.43	981	396
		265	258
707	0.8	1142	60
		56	244
828	0.71	1447	15
		127	174

Table 6.3: Sample starts obtained using various seeds while splitting the dataset at problem level for the task of binary algorithm classification on non-math and math algorithms.

6.2 Multi Label Decision Tree Classifier

Using a similar setup as in the one established in the previous section, splitting the pool of problems into a training/testing dataset, we reran the experiment, this time aiming for obtaining a model, capable of performing multi label classification.

We recorded the results after performing the task of algorithm classification, for the testing datasets, and analyzed it through the precision, recall and f1-score metrics. We limited our results to problems that have a relevant number of entries in the dataset: *strings*, *implementation*, *greedy*, *brute force*, *dp*, *divide and conquer*, *graphs*, *binary search*, *math*, *sortings*, *shortest paths*.

Class	Precision	Recall	F-score	Support
strings	0.86	0.87	0.87	756
implementation	0.95	0.93	0.94	1387
greedy	0.78	0.79	0.78	523
brute force	0.79	0.77	0.78	311
dp	0.83	0.71	0.77	35
divide and conquer	0.91	0.65	0.75	31
graphs	0.92	0.82	0.87	83
binary search	0.91	0.65	0.75	31
math	0.91	0.85	0.88	301
sortings	0.66	0.66	0.66	176
shortest paths	0.92	0.82	0.87	83
micro avg	0.87	0.85	0.86	3717
macro avg	0.86	0.77	0.81	3717
weighted avg	0.87	0.85	0.86	3717
samples avg	0.89	0.88	0.88	3717

Table 6.4: A classification report ran against the testing dataset, for the problem of algorithm classification, using a **Decision Tree** classifier and *r-Complexity* embeddings. Full report in **Appendix F**.

Results are especially good for problems that have a large support of problems, but overall

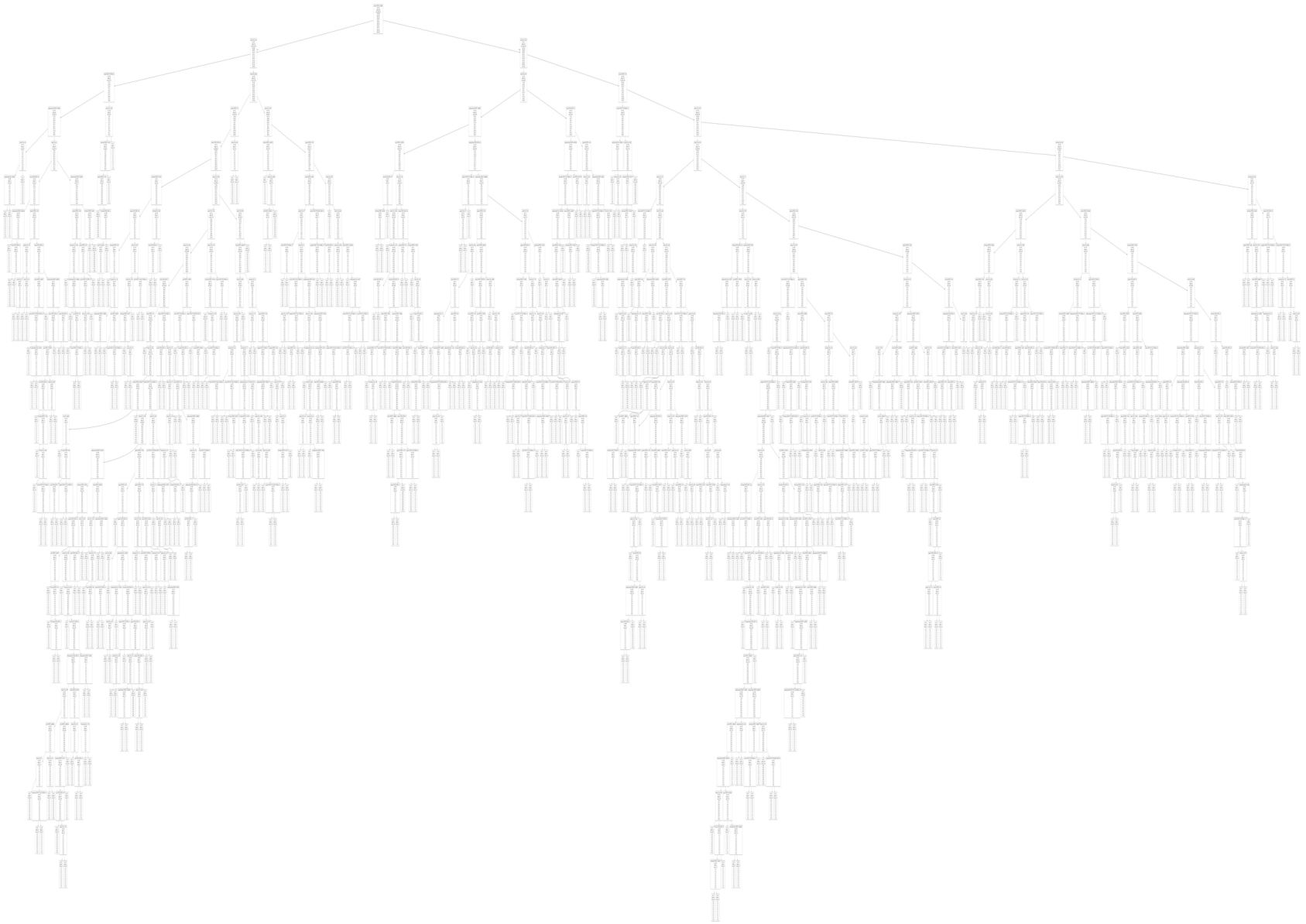


Figure 29: A graphical representation of the obtained decision tree for the task of multi label classification. Full scale image available: `https://github.com/raresraf/AlgoRAF/blob/master/prototype/dtree/multilabel_dtree_render.png`

scores are satisfying for all the analyzed classes. The training classifications performed on the data available in the training data set, were really close to reaching 100% accuracy/f1-score. Full report can be consulted in **Appendix F** of this paper.

For training such a model, we have used the scikit-learn [18] library implementation in Python, with build-in support for building a decision tree classifier. The model can be customized for various parameters. To begin with, the user can specify the function to measure the quality of a split; We have used the Gini impurity as a metric to optimize in our model. The Gini impurity has the formal definition of:

$$G = \sum_{i=1}^{C} p(i) \cdot (1 - p(i))$$

where C represents the number of classes and $p(i)$ is the probability of picking a entry from the class i.

```
# Importing the tree package.
from sklearn import tree

# Initialising a default DecisionTreeClassifier.
model = tree.DecisionTreeClassifier()

# Training a Decision Tree Classifier using the train dataset.
model.fit(train_dataset_features, train_dataset_labels)

# Evaluate the model on the dataset.
test_predicted_labels = model.predict(test_dataset_features)

# Build the classification report using the model prediction and the
   ground truth labels.
from sklearn.metrics import classification_report
classification_report(test_dataset_labels, test_predicted_labels)
```

Listing 6.1: Snippet for training and testing a DecisionTreeClassifier using scikit-learn library.

6.3 Multi Label Random Forest Classifier

Similar to the decision tree classifier, our experiments show-cased slight improvements when changing the model to a random forest classifier. This model has yielded overall better results in most of the cases than a simple decision tree. Different than the classic decision tree classifier, a random forest performs the classification using meta estimators. Those estimators fit a variety of more simple decision tree classifiers, for samples of the training dataset.

The final prediction is computed as an average for the output of the classifiers, which might improve the accuracy, as well as controlling over-fitting the model. For measuring the quality of a split, we have used the Gini impurity.

Class	Precision	Recall	F-score	Support
strings	0.93	0.84	0.88	756
implementation	0.93	0.97	0.95	1387
greedy	0.91	0.65	0.76	523
brute force	1.00	0.66	0.79	311
dp	0.87	0.77	0.82	35
divide and conquer	1.00	0.71	0.83	31
graphs	0.84	0.83	0.84	83
binary search	1.00	0.71	0.83	31
math	0.95	0.81	0.88	301
sortings	0.96	0.42	0.58	176
shortest paths	0.84	0.83	0.84	83
micro avg	0.93	0.82	0.87	3717
macro avg	0.93	0.74	0.82	3717
weighted avg	0.93	0.82	0.86	3717
samples avg	0.91	0.86	0.87	3717

Table 6.5: A classification report ran against the same testing dataset from the report in the previous section, for the problem of algorithm classification, using a **Random Forest** classifier and *r-Complexity* embeddings. Overall, the results obtaining this ensemble model have been better than by using a single, simple, decision tree classifier, for almost all criterias and analyzed metrics. Full report in **Appendix G**.

```
# Import the ensemble package.
from sklearn import ensemble

# Initialising a default RandomForestClassifier.
model = ensamble.RandomForestClassifier()

# Training a Random Forest Classifier using the train dataset.
model.fit(train_dataset_features, train_dataset_labels)

# Evaluate the model on the dataset.
test_predicted_labels = model.predict(test_dataset_features)

# Build the classification report using the model prediction and the
    ground truth labels.
from sklearn.metrics import classification_report
classification_report(test_dataset_labels, test_predicted_labels)
```

Listing 6.2: Snippet for training and testing a RandomForestClassifier using the scikit-learn library.

6.4 XGBoost (eXtreme Gradient Boosting)

XGBoost is a classifier build on top of the gradient boosted decision trees technique. In a similar setup, we have benchmark the performances of a XGBoost classifier against the problem of algorithm classification. In our research, this method has showcased the **best performance**, in terms of *precision*, *recall* and *F1-score* for the analyzed scenarios.

Class	Precision	Recall	F-score	Support
strings	**0.94**	**0.9**	**0.92**	756
implementation	0.94	**0.98**	**0.96**	1387
greedy	**0.92**	0.77	**0.84**	523
brute force	0.98	0.77	**0.86**	311
dp	**0.87**	0.74	0.8	35
divide and conquer	**1.0**	0.68	0.81	31
graphs	0.91	**0.88**	**0.9**	83
binary search	**1.0**	0.68	0.81	31
math	**0.97**	**0.91**	**0.94**	301
sortings	0.95	**0.61**	**0.74**	176
shortest paths	0.91	**0.88**	**0.9**	83
micro avg	**0.94**	**0.88**	**0.91**	3717
macro avg	**0.94**	**0.8**	**0.86**	3717
weighted avg	**0.94**	**0.88**	**0.91**	3717
avg	**0.94**	**0.91**	**0.91**	3717

Table 6.6: A classification report ran against the testing dataset, for the problem of algorithm classification, using the **XGBoost** classifier and *r-Complexity* embeddings. In **bold**, we captured the results that were better than the previous two models, decision tree and random forest classifiers. Full report in **Appendix H**.

```
# XGBoost required imports to be compatible with scikit-learn
import xgboost as xgb

# MultiOutputClassifier dependency.
from sklearn.multioutput import MultiOutputClassifier
```

```
6
7  # Initialising a XGBClassifier.
8  # https://xgboost.readthedocs.io/en/stable/python/python_api.html#
       module-xgboost.sklearn
9  xgb_classifier = xgb.XGBClassifier(objective='binary:logistic')
10
11 # MultiOutputClassifier strategy consists of fitting one classifier per
       target. This allows multiple target variable classifications. The
       purpose of this class is to extend estimators to be able to estimate
       a series of target functions (f1,f2,f3, ... ,fn) that are trained
       on a single X predictor matrix to predict a series of responses (y1,
       y2,y3, ... ,yn).
12 # https://scikit-learn.org/stable/modules/multiclass.html#
       multioutputclassifier
13 model = MultiOutputClassifier(xgb_classifier)
14
15 # Training a Multi Output Classifier build on multiple XGB classifiers
       using the train dataset.
16 model.fit(train_dataset_features, train_dataset_labels)
17
18 # Evaluate the model on the dataset.
19 test_predicted_labels = model.predict(test_dataset_features)
20
21 # Build the classification report using the model prediction and the
       ground truth labels.
22 classification_report(test_dataset_labels, test_predicted_labels)
```

Listing 6.3: Snippet for training and testing a XGBoost classifier combined with the MultiOutputClassifier strategy using the scikit-learn library.

6.5 Other classifiers

We have performed the same classification tasks using different models as well. We tried different setups of neural networks, such as multi-layer perceptron, but the results were unsatisfactory. We were not able to achieve more than **0.3** F1-score on the described setup with any of the architectures we experimented with.

Chapter 7

Conclusions

We have established the foundation of this research by the statement of the objective of this study. We aim to to have a better understanding of how human-written code can be automatically analyzed and what useful characteristics can be obtained from it. We have also presented various modern techniques for operating with source code and automatically generating code embeddings and we have looked into the application of automatic code understanding.

Therefore, during this work, we have developed TheCodeforcesCrawled, that is, an automatic system for building software solutions for automatic discovery of code solutions implemented and open-sourced by authors from all over the world, containing over 140905 unique sources pre-compiled for MacOS and Ubuntu. We have also worked on TheInputsCodeforces, a dataset containing a custom suite of inputs for various CodeForces problems. From a total number of 42 codeforces problems, we built 1803 synthetic problems inputs, resulting in 1110636 analysis files, containing telemetry from over 5K solutions to problems.

Calculus in big **r-Theta** notation proves to be useful in creating dynamic code embeddings. The idea behind these embeddings is simple: try to automatically provide estimations, for various metrics, the r-Theta class for the analyzed metrics of a specific algorithm (usually with unknown Bachmann–Landau Complexity). In this research, we used a simplified field of regressions, as possible versions of the generic expression as a Big r-Theta function, described generic by one of the following function:

$$\begin{cases} r \cdot log_2^p log_2(n) + X \\ r \cdot log_2^p(n) + X \\ r \cdot p^n + X, p < 1 \\ r \cdot n^p + X \\ r \cdot p^n + X, p > 1 \\ r \cdot \Gamma(n) + X \end{cases}.$$

We analyze an algorithm by the associated r-embedding. It can be built on top of any metrics collected. In particular, using the `perf` profiler, we obtained and used the following metrics in the task of algorithm classification:

- branch-misses
- branches
- context-switches
- cpu-migrations
- cycles
- instructions
- page-faults
- stalled-cycles-frontend
- task-clock

The *system* we proposed to analyse the solutions to the problems contained in the dataset can be seen as two distinct entities: **the data acquisition subsystem** and the **embeddings subsystem**. The data acquisition subsystem consists of two main parts, the **Control Plane** and the **Data Pipeline Plane**.

The method presented for transforming the source code to numerical embeddings is a helpful solution when needing to analyze the behaviour of algorithms. Using these embeddings that are based on r-Complexity [6], we have achieved very high accuracy using a simple decision tree classifier, showing the potential of this method in building an artificial understanding of the **meaning** of the **code**.

Moreover, we ran multiple experiments for other various classifiers, including but not limited to some neural networks, mainly multi-layer perceptron classifiers, random forest classifiers and XGBoost. Some of them ended improving the classification accuracy and the f1-score for the classification problem up to **94% precision, 91% recall** and **91% f1-score**.

What's next?

We presented in this paper an way of building code embeddings based on Complexity related measurements and use these telemetry as insights for the problem of algorithm classification. The greater goal of our research is to build a generic model of mapping an algorithm to a code embedding (based on the r-Complexity code embeddings technique), that can later be used for more than solving the simple problem of labeling solutions for competitive programming challenges, but for more advanced challenges too. We believe that these embeddings covers a good amount of information about what the code is really doing, and can be used to further analysis, such as but not limited to plagiarism detection, software optimisations and classification or malware detection. The core idea of moving code to embeddings gives a large degree of freedom for finding an usefulness for a particular problem. Also, tailoring the collected metrics for a certain algorithm can help reach this goal, by introducing other more relevant metrics for solving a certain task.

Appendix A

Codeforces Solutions Dataset

This appendix presents the set of problems and the corresponding number of crawled solutions (showing only those with more than 250 solutions/problem).

Problem number	Number of sources
1A	465
4A	411
71A	386
231A	374
158A	354
2A	354
263A	339
282A	328
112A	325
118A	316
236A	307
266A	301
50A	299
339A	298
281A	291
2B	289

116A	280
266B	270
69A	269
110A	265
546A	262
96A	255
58A	255
160A	253

Appendix B

Sample solution to a dynamic programming problem: 791E - Codeforces

This appendix presents a sample solution from the TheCrawlCodeforces dataset corresponding to problem **791E.CPP**. There are some notable things to consider regarding this proposed code snippet for solving this problem:

1. The competitive programming code aims to provide the source code such that computer are able to solve given problems the fastest way possible. [8] is a great resource that presents the world of competitive programming as *Given well-known Computer Science (CS) problems, solve them as quickly as possible*.
2. The above definition does not imply that solve them as quickly as possible implies trivial logic. In fact, the more advanced the techniques used, the better the outcome might be. Dynamic Programming can be easily replaced by easier to understand methods, such as simple recursions, with the cost of complexity increase.
3. In competitive programming, a working code is almost never[1] refactored. If it works, the programmer will rather use his/her time focusing on other problems.
4. The indentation is not always consistent in such snippets.
5. The comments are untrustworthy or non-existent in such snippets. However, it's common to see code commented, which is either obsolete, non-working, to be revisited, debugging tooling or optimisation ideas.
6. Consistent name scheme is not a thing. camelCase and snake_case notations may appear simultaneously.
7. Code might be repeated, which would save some time factoring the common part in a function for the programmer, but at the cost of losing the readability.
8. Long lines of code are everywhere!

[1] A little exaggeration to point out, but refactoring the code in a way that it's more readable in competitions brings no benefit, rather than losing precious time during the competition

9. Long methods are everywhere!
10. Dead/unused code is everywhere!

```
#include <bits/stdc++.h>
#define ID if (0)
using namespace std;
const int INF = 1e9;
int N, a[80], DP[76][76][76][2];
char A[80];
vector<int> pos[3];
// DP[i][j][k][last one is V ?] = minimal number of operations needed
// to make length i + j + k string, with i V's , j K's and k X's, st
// the last character is V or not.

int main() {
  cin >> N;
  cin >> (A + 1);
  vector<int> cnt(3);
  for (int i = 1; i <= N; ++i) {
    if (A[i] == 'V')
      a[i] = 0;
    else if (A[i] == 'K')
      a[i] = 1;
    else
      a[i] = 2;
    pos[a[i]].push_back(i);
    ++cnt[a[i]];
  }
[truncated]
  DP[0][0][0][0] = 0;
  for (int len = 0; len < N; ++len) {
    for (int i = 0; i <= cnt[0]; ++i) {
      for (int j = 0; j <= cnt[1]; ++j) {
        for (int k = 0; k <= cnt[2]; ++k) {
          if (i + j + k < len)
            continue;
          if (i + j + k > len)
            break;
          for (int f : {0, 1}) {
            if (DP[i][j][k][f] >= INF)
              continue;
            ID printf("DP[%d][%d][%d][%d] = %d\n", i, j, k, f, DP[i][j][k][
    f]);
            // if we append V
            if (i < cnt[0]) {
              int p = pos[0][i];
              vector<int> ccnt(3);
              for (int x = 1; x <= p; ++x) {
                ++ccnt[a[x]];
              }
```

```
47                DP[i + 1][j][k][1] =
48                    min(DP[i + 1][j][k][1], DP[i][j][k][f] + max(0, ccnt[1] -
      j) +
49                                                max(0, ccnt[2] - k));
50                ID printf("update DP[%d][%d][%d][%d] = %d\n", i + 1, j, k, 1,
51                          DP[i + 1][j][k][1]);
52              }
53 [truncated]
54            }
55          }
56        }
57      }
58    }
59    ID for (int f : {0, 1}) printf("DP[%d][%d][%d][%d] = %d\n", cnt[0], cnt
      [1],
60                                   cnt[2], f, DP[cnt[0]][cnt[1]][cnt[2]][f]);
61    cout << min(DP[cnt[0]][cnt[1]][cnt[2]][0], DP[cnt[0]][cnt[1]][cnt[2]][1])
62         << endl;
63 }
64
65 /*#include <bits/stdc++.h>
66 #define ID if(0)
67 using namespace std;
68 const int INF = 1e9;
69 int N;
70 int DP[80][80][3][3];
71 int a[80];
72 char A[80];
73 // ... 0 , 1 ... is not allowed
74 int main(){
75     cin >> N;
76     cin >> (A + 1);
77     for(int i = 1; i <= N; ++i){
78         if(A[i] == 'V')a[i] = 0;
79         else if(A[i] == 'K')a[i] = 1;
80         else a[i] = 2;
81     }
82     for(int l = 1; l <= N; ++l){
83         for(int r = l; r <= N; ++r){
84             for(int x : {0 , 1 , 2}){
85                 for(int y : {0 , 1 , 2}){
86                     DP[l][r][x][y] = INF;
87                 }
88             }
89         }
90     }
91 [truncated]
92 */
```

Appendix C

Full content of problem: 670A - Codeforces

This appendix presents a sample problem (670A), referenced in this material, from the Codeforces dataset.

A. Holidays

```
time limit per test: 1 second
memory limit per test: 256 megabytes
input: standard input
output: standard output
```

On the planet Mars a year lasts exactly n days (there are no leap years on Mars). But Martians have the same weeks as earthlings — 5 work days and then 2 days off. Your task is to determine the minimum possible and the maximum possible number of days off per year on Mars.

Input: The first line of the input contains a positive integer n ($1 \leq n \leq 1000000$) — the number of days in a year on Mars.

Output: Print two integers — the minimum possible and the maximum possible number of days off per year on Mars.

Examples:

input(1)

```
14
```

output(1)

```
4 4
```

input(2)

```
2
```

output(2)

```
0 2
```

Note:

In the first sample there are 14 days in a year on Mars, and therefore independently of the day a year starts with there will be exactly 4 days off .

In the second sample there are only 2 days in a year on Mars, and they can both be either work days or days off.

Appendix D

Sample embedding generated by the system for a solution to the problem: 1366A - Codeforces

This appendix presents a embedding generated by the system for a solution to the problem **1366A**, referenced in this material, from the Codeforces dataset.


```
{
    "path": "../TheOutputsCodeforces/processed/atomic_perf/
  results_code/
    AKarunakaran/CodeForces/1366A/AGGREGATED.RAF",
    "metrics": {
        "branch-misses": {
            "FEATURE_CONFIG": 1,
            "FEATURE_TYPE": "POLYNOMIAL",
            "INTERCEPT": 12539.832752136186,
            "R-VAL": 1.6796273106352073
        },
        "branches": {
            "FEATURE_CONFIG": 1,
            "FEATURE_TYPE": "POLYNOMIAL",
            "INTERCEPT": 358889.9768014151,
            "R-VAL": 387.55150433699623
        },
        "context-switches": {
            "FEATURE_CONFIG": 0,
            "FEATURE_TYPE": "LOG_POLYNOMIAL",
            "INTERCEPT": 0.0,
            "R-VAL": 0.0
```

```
        },
        "cpu-migrations": {
            "FEATURE_CONFIG": 0,
            "FEATURE_TYPE": "LOG_POLYNOMIAL",
            "INTERCEPT": 0.0,
            "R-VAL": 0.0
        },
        "cycles": {
            "FEATURE_CONFIG": 1,
            "FEATURE_TYPE": "POLYNOMIAL",
            "INTERCEPT": 1696753.219927505,
            "R-VAL": 889.2213912893395
        },
        "instructions": {
            "FEATURE_CONFIG": 1,
            "FEATURE_TYPE": "POLYNOMIAL",
            "INTERCEPT": 2226336.1339648864,
            "R-VAL": 1860.1563658155937
        },
        "page-faults": {
            "FEATURE_CONFIG": 0,
            "FEATURE_TYPE": "LOG_POLYNOMIAL",
            "INTERCEPT": 124.02490265239256,
            "R-VAL": -0.09540014361945973
        },
        "stalled-cycles-frontend": {
            "FEATURE_CONFIG": 1,
            "FEATURE_TYPE": "POLYNOMIAL",
            "INTERCEPT": 2322967.300115998,
            "R-VAL": 1671.3589054117876
        },
        "task-clock": {
            "FEATURE_CONFIG": 1,
            "FEATURE_TYPE": "POLYNOMIAL",
            "INTERCEPT": 2.456778717166007,
            "R-VAL": 0.0018171698788728317
        }
    }
}
```

Appendix E

Sample math problem: 546A - Codeforces

This appendix presents a sample problem (546A), referenced in this material, from the Codeforces dataset.

A. Soldier and Bananas

```
time limit per test: 1 second
memory limit per test: 256 megabytes
input: standard input
output: standard output
```

A soldier wants to buy w bananas in the shop. He has to pay k dollars for the first banana, $2k$ dollars for the second one and so on (in other words, he has to pay $i \cdot k$ dollars for the i-th banana).

He has n dollars. How many dollars does he have to borrow from his friend soldier to buy w bananas?

Appendix F

Full training and testing results - Multi Label Decision Tree

The below represents the classification report obtained during our experiments, using a multi label decision tree trained on the training dataset, and then evaluated on the same data, already existent in the training dataset:

	precision	recall	f1-score	support
strings	1.00	1.00	1.00	1522
implementation	1.00	1.00	1.00	2788
greedy	1.00	1.00	1.00	1060
brute force	1.00	1.00	1.00	645
dp	1.00	1.00	1.00	72
divide and conquer	1.00	1.00	1.00	67
graphs	1.00	1.00	1.00	159
binary search	1.00	1.00	1.00	67
math	1.00	1.00	1.00	711
sortings	1.00	1.00	1.00	334
shortest paths	1.00	1.00	1.00	159
micro avg	1.00	1.00	1.00	7584
macro avg	1.00	1.00	1.00	7584
weighted avg	1.00	1.00	1.00	7584
samples avg	0.99	0.99	0.99	7584

The below represents the classification report obtained during our experiments, using a multi label decision tree trained on the training dataset, and then evaluated on a separate set of data from the initially data set, which has not been used during test, which we called testing dataset:

	precision	recall	f1-score	support
strings	0.86	0.87	0.87	756
implementation	0.95	0.93	0.94	1387
greedy	0.78	0.79	0.78	523
brute force	0.79	0.77	0.78	311
dp	0.83	0.71	0.77	35
divide and conquer	0.91	0.65	0.75	31
graphs	0.92	0.82	0.87	83
binary search	0.91	0.65	0.75	31
math	0.91	0.85	0.88	301
sortings	0.66	0.66	0.66	176
shortest paths	0.92	0.82	0.87	83
micro avg	0.87	0.85	0.86	3717
macro avg	0.86	0.77	0.81	3717
weighted avg	0.87	0.85	0.86	3717
samples avg	0.89	0.88	0.88	3717

Results reproducible in the open-source repository [1].

[1] https://github.com/raresraf/AlgoRAF/blob/master/prototype/dtree/MultiLabelDecisionTree.ipynb

Appendix G

Full training and testing results - Multi Label Random Forest

The below represents the classification report obtained during our experiments, using a multi label random forest model trained on the training dataset, and then evaluated on the same data, already existent in the training dataset:

	precision	recall	f1-score	support
strings	1.00	1.00	1.00	1522
implementation	1.00	1.00	1.00	2788
greedy	1.00	0.99	0.99	1060
brute force	1.00	0.99	0.99	645
dp	1.00	1.00	1.00	72
divide and conquer	1.00	0.99	0.99	67
graphs	1.00	0.99	0.99	159
binary search	1.00	0.99	0.99	67
math	1.00	0.99	1.00	711
sortings	1.00	0.97	0.98	334
shortest paths	1.00	0.99	0.99	159
micro avg	1.00	0.99	1.00	7584
macro avg	1.00	0.99	0.99	7584
weighted avg	1.00	0.99	1.00	7584
samples avg	0.99	0.99	0.99	7584

The below represents the classification report obtained during our experiments, using a multi label random forest model trained on the training dataset, and then evaluated on a separate set of data from the initially data set, which has not been used during test, which we called testing dataset:

	precision	recall	f1-score	support
strings	0.93	0.84	0.88	756
implementation	0.93	0.97	0.95	1387
greedy	0.91	0.65	0.76	523
brute force	1.00	0.66	0.79	311
dp	0.87	0.77	0.82	35
divide and conquer	1.00	0.71	0.83	31
graphs	0.84	0.83	0.84	83
binary search	1.00	0.71	0.83	31
math	0.95	0.81	0.88	301
sortings	0.96	0.42	0.58	176
shortest paths	0.84	0.83	0.84	83
micro avg	0.93	0.82	0.87	3717
macro avg	0.93	0.74	0.82	3717
weighted avg	0.93	0.82	0.86	3717
samples avg	0.91	0.86	0.87	3717

Results reproducible in the open-source repository [1].

[1] https://github.com/raresraf/AlgoRAF/blob/master/prototype/dtree/MultiLabelRandomForest.ipynb

Appendix H

Full training and testing results - XGBoost classifier

The below represents the classification report obtained during our experiments, using a XGBoost model trained on the training dataset, and then evaluated on the same data, already existent in the training dataset:

	precision	recall	f1-score	support
strings	1.00	1.00	1.00	1522
implementation	1.00	1.00	1.00	2788
greedy	1.00	1.00	1.00	1060
brute force	1.00	1.00	1.00	645
dp	1.00	1.00	1.00	72
divide and conquer	1.00	1.00	1.00	67
graphs	1.00	1.00	1.00	159
binary search	1.00	1.00	1.00	67
math	1.00	1.00	1.00	711
sortings	1.00	1.00	1.00	334
shortest paths	1.00	1.00	1.00	159
micro avg	1.00	1.00	1.00	7584
macro avg	1.00	1.00	1.00	7584
weighted avg	1.00	1.00	1.00	7584
samples avg	0.99	0.99	0.99	7584

The below represents the classification report obtained during our experiments, using a multi label random forest model trained on the training dataset, and then evaluated on a separate set of data from the initially data set, which has not been used during test, which we called testing dataset:

	precision	recall	f1-score	support
strings	0.94	0.90	0.92	756
implementation	0.94	0.98	0.96	1387
greedy	0.92	0.77	0.84	523
brute force	0.98	0.77	0.86	311
dp	0.87	0.74	0.80	35
divide and conquer	1.00	0.68	0.81	31
graphs	0.91	0.88	0.90	83
binary search	1.00	0.68	0.81	31
math	0.97	0.91	0.94	301
sortings	0.95	0.61	0.74	176
shortest paths	0.91	0.88	0.90	83
micro avg	0.94	0.88	0.91	3717
macro avg	0.94	0.80	0.86	3717
weighted avg	0.94	0.88	0.91	3717
samples avg	0.94	0.91	0.91	3717

Results reproducible in the open-source repository [1].

[1] https://github.com/raresraf/AlgoRAF/blob/master/prototype/xgboost.ipynb

Bibliography

[1] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. code2seq: Generating sequences from structured representations of code. *arXiv preprint arXiv:1808.01400*, 2018.

[2] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–29, 2019.

[3] Alexandru Calotoiu. *Automatic Empirical Performance Modeling of Parallel Programs*. PhD thesis, Technische Universität, 2018.

[4] Hanjun Dai, Bo Dai, and Le Song. Discriminative embeddings of latent variable models for structured data. In *International conference on machine learning*, pages 2702–2711, 2016.

[5] Karel Driesen and Urs Hölzle. Limits of indirect branch prediction. *Santa Barbara, CA, USA, Tech. Rep*, 1997.

[6] Rares Folea and Emil-Ioan Slusanschi. A new metric for evaluating the performance and complexity of computer programs: A new approach to the traditional ways of measuring the complexity of algorithms and estimating running times. In *2021 23rd International Conference on Control Systems and Computer Science (CSCS)*, pages 157–164. IEEE, 2021.

[7] Klaus Greff, Rupesh K Srivastava, Jan Koutník, Bas R Steunebrink, and Jürgen Schmidhuber. Lstm: A search space odyssey. *IEEE transactions on neural networks and learning systems*, 28(10):2222–2232, 2016.

[8] Steven Halim, Felix Halim, Steven S Skiena, and Miguel A Revilla. *Competitive programming 3*. Citeseer, 2013.

[9] Scott W Haney. Is c++ fast enough for scientific computing? *Computers in Physics*, 8(6):690–696, 1994.

[10] Simon Haykin. Self-organizing maps. *Neural networks-A comprehensive foundation, 2nd edition, Prentice-Hall*, 1999.

[11] Radu Cristian Alexandru Iacob, Vlad Cristian Monea, Dan Rădulescu, Andrei-Florin Ceapă, Traian Rebedea, and Ștefan Trăușan-Matu. Algolabel: A large dataset for multi-label classification of algorithmic challenges. *Mathematics*, 8(11):1995, 2020.

[12] Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. Learning and evaluating contextual embedding of source code. In Hal Daumé III and Aarti Singh, editors, *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 5110–5121. PMLR, 13–18 Jul 2020.

[13] Kimmo Kiviluoto. Topology preservation in self-organizing maps. In *Proceedings of International Conference on Neural Networks (ICNN'96)*, volume 1, pages 294–299. IEEE, 1996.

[14] Teuvo Kohonen. Self-organizing maps: ophmization approaches. In *Artificial neural networks*, pages 981–990. Elsevier, 1991.

[15] Chuanpeng Li, Chen Ding, and Kai Shen. Quantifying the cost of context switch. In *Proceedings of the 2007 workshop on Experimental computer science*, pages 2–es, 2007.

[16] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[17] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. *Advances in neural information processing systems*, 26:3111–3119, 2013.

[18] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[19] Folea Rares. *Metrics for evaluating the performance and complexity of computer programs*. UPB, 2020.

[20] Maja Rudolph and David Blei. Dynamic embeddings for language evolution. In *Proceedings of the 2018 World Wide Web Conference*, pages 1003–1011, 2018.

[21] Stef van den Elzen and Jarke J. van Wijk. Baobabview: Interactive construction and analysis of decision trees. In *2011 IEEE Conference on Visual Analytics Science and Technology (VAST)*, pages 151–160, 2011.

[22] Chr Von der Malsburg. Self-organization of orientation sensitive cells in the striate cortex. *Kybernetik*, 14(2):85–100, 1973.

[23] Wikipedia contributors. Time complexity — Wikipedia, the free encyclopedia, 2021. [Online; accessed 18-April-2021].

[24] David S Wile. Abstract syntax from concrete syntax. In *Proceedings of the 19th international conference on Software engineering*, pages 472–480, 1997.

[25] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. Neural network-based graph embedding for cross-platform binary code similarity detection. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 363–376, 2017.