



UNIVERSIDAD NACIONAL DE COLOMBIA

Automatic source code analysis for assessment of solutions to programming tasks

Lina Fernanda Rosales Castro

Universidad Nacional de Colombia
Facultad de Ingeniería
Bogotá, Colombia
2019

Automatic source code analysis for assessment of solutions to programming tasks

Lina Fernanda Rosales Castro

Thesis presented as partial requirement to obtain the degree of:
Magíster en Ingeniería de Sistemas y computación

Director:

Ph.D. Fabio A. González O.

Co-Director:

Ph.D. Felipe Restrepo Calle

Research Line:

Intelligent systems and Programming languages

Research group:

MindLAB - PLaS

Universidad Nacional de Colombia
Faculty of Engineering, Department of Systems and Industrial Engineering
Bogotá, Colombia
2019

Acknowledgments

To my family who supported me, friends who encouraged me, teachers from Universidad Nacional de Colombia who inspired me and all the amazing people that, somehow, were with me along this path.

Yay!

Resumen

Calificar código fuente es una tarea que cada instructor de programación debe enfrentar en su día a día. Esta tarea conlleva varios retos entre ellos la cantidad de código a calificar, la dificultad de la tarea propuesta, entender cómo cada estudiante está intentando resolver la tarea y proveer realimentación para garantizar una aprendizaje de calidad. Esta tarea no solo es relevante en contextos académicos, por ejemplo en la industria, calificar efectivamente código fuente es crucial en el proceso para contratar al programador adecuado para una vacante. En este trabajo, proponemos una modificación al proceso regular de calificar código agregando un componente automático (o por lo menos semi-automático) para ayudar a un instructor de programación en esta tarea. Nuestra propuesta incluye una rúbrica para calificar código fuente, la construcción de un data set, la extracción y el análisis del código; para finalmente mostrar nuestra solución al problema de calificar código fuente de manera automática resolviendo la tarea como un problema de clasificación.

Palabras clave: Calificar código fuente, análisis de código fuente, rúbricas .

Abstract

Grading source code is a task that every programming instructor has to face in her daily life. This task has many challenges among them the amount of source code the instructor has to grade, the difficulty of the proposed problems, understanding how every student is trying to approach the solution, and providing good feedback to guarantee quality learning. This task is also relevant on hiring processes, to assure the contractor party is effectively hiring the best developer for the job. In this work we are proposing an enhancement of the regular grading process in an academic context to grade source code in an automatic (or at least semi-automatic) way to help out the instructor in this process. We propose a rubric to grade source code that can be easily used to evaluate several solutions to a programming problem, then we show the construction of the data set, the feature extraction and analysis of source code, and finally our results for grading source code framing the task as a classification problem.

Key words: Grade source code, source code analysis, rubric to grade source code .

Content

Acknowledgments	v
Resumen	vii
Abstract	vii
1 Introduction	1
1.1 Motivation	1
1.2 Problem identification	2
1.3 Objectives	3
1.4 Contributions	3
1.5 Thesis structure	4
2 Related works	5
2.1 Rubrics to grade source code	5
2.2 Source code analysis	8
2.3 Automatic assessment of source code	10
3 A method for assessing source code using machine learning and source code analysis techniques	13
3.1 Source code rubric	14
3.1.1 Building the rubric	15
3.1.2 Levels of the rubric	15
3.2 Grader	19
3.2.1 Preprocess of the data	19
3.2.2 Feature extraction	20
3.2.3 Training	20
3.2.4 Classification task	23
4 Experimental evaluation	24
4.1 Experimental setup	24
4.1.1 Dataset	24
4.1.2 Tagging the dataset	31
4.1.3 Classification models	32

4.2	Experimental results	35
4.2.1	Support Vector Machine Lineal (SVML)	38
4.2.2	Random Forest (RF)	38
4.2.3	Discussion	41
5	Conclusions and Future Work	45
5.1	Conclusions	45
5.2	Future Work	46
	Bibliography	48

List of figures

3-1. Proposed method	14
3-2. An example of a source code (green) and the tokens extracted using ANTLR (orange)	20
3-3. Histogram of tokens appearance for 451 samples of source code examples solving a punctual task	22
3-4. A list of the most frequent (green) n-grams and less frequent (orange) n-grams for a set of 451 samples of source code solving a punctual task	22
4-1. Distribution of the problems per tag	27
4-2. Distribution of the problems with the tag <i>dfs</i> and <i>similar</i>	28
4-3. An early visualization of the submissions, in green the submissions with tag <i>math</i> , in blue the submissions with the tag <i>graphs</i> and in in red the submissions belonging to the tag <i>dfs and similar</i>	30
4-4. Distribution among the verdicts and problems	31
4-5. User interface in UNCode displaying the proposed rubric	32
4-6. User interface in UNCode used to label the dataset	33
4-7. SVM L parameters exploration with several representations	35
4-8. Random forest parameters exploration with concatenation of all representations.	36
4-9. Random forest parameters exploration with features extracted from AST representation.	36
4-10. Random forest parameters exploration with features extracted from the source code as text.	37
4-11. SVM L confusion Matrix from the classification using the concatenation of features extracted from the AST, n-grams as text and token counting.	39
4-12. SVM L confusion Matrix from the classification using features extracted from the AST.	39
4-13. SVM L confusion Matrix from the classification using n-grams features using source code as text.	40
4-14. SVM L confusion Matrix from the classification using counting tokens as classification.	40
4-15. RF confusion Matrix from the classification using the concatenation of features extracted from the AST, n-grams as text and token counting.	42

4-16. RF confusion Matrix from the classification using features extracted from the AST.	42
4-17. RF confusion Matrix from the classification using n-grams features using source code as text.	43
4-18. RF confusion Matrix from the classification using counting tokens as classification.	43

List of Tables

2-1. Rubric proposed in Aspiring Minds to grade source code [32].	6
2-2. Rubric proposed by the authors in [5] to grade Java Source Code.	7
3-1. Description of the level one in the rubric	16
3-2. Description of the level two in the rubric	16
3-3. Description of the level three in the rubric	17
3-4. Description of the level four in the rubric	18
3-5. Description of the level five in the rubric	18
3-6. Top 10 of features appearance in a sample source code	21
3-7. Source code features extracted from the AST	21
4-1. User ranking in Codeforces	25
4-2. Number of submissions per tag in Codeforces	26
4-3. Number of submissions with tag dfs and similar and tags related	29
4-4. Proportions of the verdicts per problems	29
4-5. Distribution of the submissions per grade tagged in the data set.	32
4-6. Best value of the parameter C for every representation	34
4-7. Recap of parameters exploration for RF	35
4-8. Summary of results with the classifiers SVMML and Random Forest and several source code representations.	37

1 Introduction

Having large amounts of data and extracting information from them is a very common problem in different fields and source code is not the exception. Considering source code in an academic context, the information extracted could be used to provide feedback in different ways. An important and useful way is to provide an assessment, so that the students can benefit from early disambiguation of misconceptions [24].

The assessment, interpreted as an objective quantification of how good a source code is solving a particular task, could give to the instructor an idea of how students are acquiring and assimilating the knowledge [24]. But not only instructors benefit from it: any reviewer who aims to grade source code, for example in a hiring process, can use it as part of the assessment to highlight aspects of the candidate during a coding interview [32]; this problem is relevant in terms of the increasing number of programmers that the industry requires [9] and the short amount of time a recruiter has to filter the best candidates for the job [32]. An objective assessment could also be useful for the task of ranking the performance of a programmer, identifying her strengths, and in what kind of field her skills are better [32]. But providing an objective assessment implies a big effort for the reviser or instructor, and in order to decrease her work [3] it is important to do it automatically, or at least, semi-automatically.

1.1. Motivation

Good feedback is one of the central keys for quality in students learning [24]. Feedback can be understood as any kind of communication between the instructor and the students. It can be guidance, an email, or a comment during class, among others [24]. But the main characteristic is that it comes from the instructor's understanding of how the student solved (or tried to solve) an exercise, or in this case, of how the programmer is trying to solve a problem.

In [13] is mentioned that the most powerful feedback occurs when feedback is given to the instructor: how well they have taught, who they taught well, what they have or have not taught well. Here, the importance of also give feedback to the instructor is shown as she can act as an effective replicator of the feedback.

Automatic source code assessment has a wide amount of related work and has been investigated since the 60s [17, 18]. Its aim is to reduce the amount of work that an evaluator does [32], and improve the speed, availability, consistency, and objectivity of the assessment [2]. All of these improvements could be applied to source code from students, but also from candidates in a hiring process, and as said in [5], besides improving the feedback, it expands the pool of good candidates.

Having a large amount of source code to be assessed is a problem for supervisors aiming to give feedback. It becomes more difficult as the number of tasks the instructors have to assess increases; it is difficult to do it by hand, but doing it automatically is a challenge. To provide an objective assessment implies determining the level of expertise of the coder, and to determine a scale to measure how good the coder is at solving a given programming problem. Also, it is a challenge to determine how to classify the coder objectively.

1.2. Problem identification

Grading source code requires a way to assess it objectively, and some authors have proposed rubrics as a way to do it. Authors [22] have proposed generic rubrics to assess source code, while others [32, 5] [] have proposed more specific rubrics to grade source code taking into account characteristics extracted from it. These characteristics could be source-code-based or text-based (using the source code as plain text). Among these characteristics, several authors use one or the other, and even both combined [26, 19, 21]. Even so, these rubrics are lack of clear boundaries, meaning that a given piece of source code graded with them could have multiple grades and also it is difficult to select only one grade due to the ambiguity of the description, also a rubric general enough to be used in several problems is still missing. Regarding the source code representation, there is still a wide spectrum of features than can be extracted from the source code and explored to achieve the automatic grading task.

There are some research questions that will be answered during this work:

- What features from source code in terms of syntax, semantic and software metrics are more useful to give automatic assessment of source code?
- What advantages does an automatic assessment using features extracted from code itself (like syntax, semantic and software metrics) provide over a binary approach in the task of automatic assessment?
- How does an automatic assessment based on machine learning and source code analysis compare to one given by an instructor?

- How could a source code assessment process be improved by the use of source code features, source code analysis and machine learning?

1.3. Objectives

The general objective of this work is to develop a method to automatically or semi automatically assess the source code of the solutions of different programming tasks using machine learning and source code analysis techniques. To accomplish this general objective, five specific objectives were defined:

- To adapt an objective evaluation criteria for source code assessment.
- To build a data set composed of source code solutions for different programming problems and annotate the data set with the evaluation criteria.
- To design or adapt a method for the automatic (or semiautomatic) assessment of source code.
- To implement the proposed method for the automatic (or semiautomatic) assessment of source code.
- To evaluate the proposed method over the task of source code assessment.

1.4. Contributions

The main contributions of this work are:

- An early published article on visualization of source code from students, including an exploration on how to represent it [30].
- A method to include an automatic model in a regular grading academic process for the assessment of source code of different programming tasks.
- A generic rubric to grade source code that could be used with any type of programming problems.
- Scrapped submissions from Codeforces¹ from the first active contest until July 1st 2017.
- A labeled data set using the rubric with submissions selected from Codeforces problems with the *dfs* (*Depth-first search*) and *similar* tag.

¹<https://codeforces.com/>

- A software module in UNCode [27] to manually grade source code based on the proposed rubric (or another one if it is required).
- An exploration of source code representation and features for the task of grading source code.
- The results of using the proposed model to grade the data set represented with the explored features.

1.5. Thesis structure

The document is structured as follows: Section 2 describes the related work in the field; Section 3 describes the methodology used, a description of the proposed rubric, how it was constructed and the information flow in the proposed solution; Section 4 describes the experimental evaluation, including the dataset used, the experimental setup and the discussion of the results. Finally, Section 5 describes the conclusions and future works.

2 Related works

The aim of this chapter is to provide context on the work that has been done in three main areas: rubrics to grade source code, source code analysis, and automatic assessment of source code.

2.1. Rubrics to grade source code

Accordingly to [16], in a learning context, a rubric is a detailed description of the most important aspects a student should accomplish to succeed on their learning, as well as what an instructor promises to teach students. Besides, they provide the instructors with a consistent way of measuring the expectations from a learner in a given topic [4] (CARLA - Center for Advanced Research on Language Acquisition).

Rubrics can be classified in two main categories [4]:

- Holistic rubrics: This kind of rubrics evaluates the overall performance of the learner. Several quantitative and qualitative aspects are taken into account.
- Analytic rubrics: In this kind of rubrics, the performance of the learner is evaluated in separate categories and rated individually.

When grading source code, modeled as evaluating an open-response assessment, the problem of determining how good or how close a given implementation of a problem is from a correct well known implementation is crucial. According to [1], a rubric is crucial because we are trying to measure how close a given solution is from a correct solution. A rubric maps the source code to a given level of expertise, defined by the rubric, of the person solving the problem. For example, accordingly to [32], a high-level in the rubric could mean the given source code meets a set of given and well defined criteria on correctness and efficiency, on the other side, a low level in the rubric could mean the opposite, the solution is far away from a correct solution.

In [32] a rubric to grade source code is proposed based on deconstructing how an expert would grade source code. They propose a set of features which, they considered, experts look for when grading, such as checking basic keywords, variables, logic structures and printed messages. They propose a rubric that grades in 5 levels, going from gibberish code to a

Table 2-1: Rubric proposed in Aspiring Minds to grade source code [32].

Level	Description
Completely correct and efficient	An efficient implementation of the problem using the right control structures, data dependencies and consideration of nuances corner conditions of the logic
Correct with silly errors	Correct control structures and critical data-dependencies incorporated. Some silly mistakes fail the code to pass test-cases.
Inconsistent logical structures	Right control structures exist with few/partially correct data dependencies.
Emerging basic structures	Appropriate keywords and tokens present, showing some understanding of a part of the problem.
Gibberish Code	Seemingly unrelated to the problem at hand.

completely correct and efficient solution, with intermediate stages of emerging basic structures, inconsistent logical structures, and correct with silly errors. It is mentioned that this rubric was made to grade code resulting from programming job interviews written in several programming languages. The proposed rubric is showed in the Table **2-1**.

Based on the rubric in the Table **2-1**, the work in [5] proposed another similar rubric, but being more specific in the description of each level and oriented to grade an specific problem they selected in their work. The aim of this rubric was to grade source code from programming tasks in academic contexts (programming courses). Beside, they mention that the rubric was made only to grade source code written in Java. Table **2-2** shows the proposed rubric.

Some other rubrics were proposed to grade source code in academic contexts but for more specific tasks. For example, the one proposed in [22] was made to grade Object Oriented Programming task (OOP), in this case the aim of the rubric was not to grade the source code itself but several skills a student should have at solving OOP tasks such as design, construction and testing of the solution. This rubric has several descriptions per skill graded, each of the descriptions, generally, comes from the disability to even understand the problem till being capable to solve the task successfully. This rubric was proposed based on the experience of the instructor on what a OOP task should have.

Table 2-2: Rubric proposed by the authors in [5] to grade Java Source Code.

Level	Description
Grade 5	Perfect solution, conforms to complexity requirements.
Grade 4	Very good solution, but is not Grade 5 because a minor mistake causes test-cases to fail.
Grade 3	A good algorithmic approach is present but complexity requirements are violated. Or the solution contains a mistake that indicates a lower level of understanding of the language or data structures. The presence of this mistake disrupts the otherwise possibly correct flow of the algorithm. The fact that the mistake remains in the submitted code also indicates that the programmer was not able to properly test the code
Grade 2	The solution contains errors that indicate a misunderstanding of the problem.
Grade 1	Barely a solution. Incomplete logic or a missing algorithm structure that could resemble a correct solution approach. It could also be a decent starting algorithm structure but flooded with so many Grade 2 mistakes that the solution can no longer compete with other Grade 2 solutions.

Other rubrics proposed also to grade source code from students that were implemented in courses. For instance, there are rubrics that grades source code in several facets like coding standards, documentation, running time and efficiency, having a grade per graded facet¹.

The rubric proposed in [12] has a slight different approach, instead of describing the characteristic a source code should have to deserve a grade, it provides a list of general characteristics, such as the file extension of the solution or specifics in how to solve the given problem, and based on how many of those characteristics a source code has, a grade is assigned.

As it has been shown, there are several rubrics proposed to grade source code, going from only grading how the coder is understanding the problem and implementing the solution or the source code itself.

2.2. Source code analysis

The applications in education for source code analysis, particularly in the teaching of programming have a wide spectrum. Some of them include finding bugs [1], plagiarism detection [14, 19, 20], duplicates detection [21], source code assessment [25, 8], and providing feedback [7], among others. The aim of this section is to cover some of these techniques and how different authors have treated the source code to accomplish these tasks.

Regarding to the source code, authors have used the source code as text or as source-code itself (raw). Related to use the source code as text, some authors [20, 26, 31, 19] have extracted features that are commonly used over text such as:

- Difference of length in submissions [20].
- Edit distance (represents each source code as a string and then measure the similarity of those strings) [20].
- Similarity over comments and string literals [20].
- Tokenization [26, 19].
- N-grams [31].
- Counting characters, words, etc [6].

All of these approaches try to extract knowledge from the source code using it as plain text using the fact that the source code could have some information regarding of its context

¹<https://course1.winona.edu/shatfield/air/Computer Programming Tusculum College.doc>

based on the structure and vocabulary used to code it. The authors in [31] use a very common technique used in Natural Language processing (NLP) such as n-grams over the source code to try to capture the dependencies that an element in the source code uses prior to its execution. Altogether, the authors in [20], beside the features extracted from source code as text, also use lexical features from the source code such as counting blocks of features, methods, counting constants and variables. They provide a list of 55 features extracted from source code and divide them in four categories:

- Basic block features: They define a basic block as straight line code sequence with no branches except the initial and the close ones. They take into account the basic blocks and the blocks surrounding each block (successors, predecessors, instructions, etc).
- Control flow graph features
- Method features
- Other characteristics: such as counting constants, static and local variables, among others.

In [6] other lexical features as the average number of parameters and nested loops are extracted. Beside these features, the authors in [6] also use features extracted from an Abstract Syntax Tree (AST). For each submission they obtain the AST by parsing the source code, then, they identify features such as the depth of the tree, the frequency of the leaves in the AST and frequency of nodes in the AST, among others. Beside the features extracted from the AST by itself, they also propose to obtain bi-grams from the AST, in addition, they provide a list of 58 possible nodes types excluding the leaves and then use Term Frequency Inverse Document Frequency (TF-IDF) to identify rare nodes. They provide a list of 9 features extracted from the AST. Other approaches, such as the one proposed in [23] use more simple features extracted from the AST like counting nodes together with parsing sub-trees of the AST.

The authors in [19], use statistical techniques from NLP to identify bugs and plagiarism in source code. Regarding to machine learning models, authors in [6] use a Random Forest (RF) classifier with 300 trees that, according to the authors, empirically provide the best trade off between accuracy and processing time to solve the task of de-anonymizing source code. On the other hand, the author in [20] use a Support Vector Machine (SVM) to solve a classification task in plagiarism detection. To grade source code, the authors in [32] also use SVM with lineal and Radial Basis Function (RBF) together with a RF to solve the task.

2.3. Automatic assessment of source code

There are two ways to perform the automatic source code assessment: dynamic and static. The dynamic approach implies the execution of the source code, and the static does not [15]. According to [2] there are 8 aspects to take into account when an automatic assessment is made, among the dynamic analysis: functionality, efficiency, and testing skills; and among the static analysis: coding style, programming error, software metrics, design, and special features. Most of the studies focus on dynamic analysis rather than static, program correctness is the most used factor to measure quality in the source code, and black-box testing is the most popular technique for dynamic testing [29].

Among the dynamic analysis, there is a wide group of tools to automatically grade source code, which are originally intended for programming contests. These tools, denominated online judges, attempt to grade a programming solution in terms of how well it performs on a set of test cases for a given problem. Uva Online Judge [28] is one of the most recognized among them. It was created to help programming contest competitors to improve their skills during competitions, but with time, people around the world started to use it for personal study. DOMjudge² is another online judge, it was developed by a community of developers around the world as an open source project. This online judge was developed to run programming contests such as ACM-ICPC.

However, most online judges may cause problems with the binary approach of correct or incorrect feedback. According to [5] the first one is when a solution pass all the test cases using *hard code*, which implies no editable data or parameters, and in terms of efficiency and good practices, this is not desired. The other case is when a source code did not pass the test cases but for *silly errors*, but the approach was correct and also the implementation; this may cause that potential good programmers lose the opportunity when applying to a job.

Regarding how to identify when a coder is solving a problem, we did a previous research [30] on how to visualise using a graph of the solutions submitted by the students. This may help to identify groups of solutions that are alike, which can help to the assessment in a way that an instructor may identify solutions marked as *incorrect* that are similar to solutions marked as *correct*, for example.

Another approach for automatic assessment, included into the static analysis, is to identify software metrics or features related to grammar or semantic over the source code [33, 14], and then, measure what those features are saying about the code. In [32], they identify features related to the source code as the number of loops, numbers of expressions containing a not equal, the number of times an operator occurs, among others.

²<https://www.domjudge.org/>

Recently, some approaches have intended to perform an automatic assessment oriented to grading coders that are applying for a job. This with the aim of reducing the time expended during the hiring, but also increasing the chances of the candidates of getting a job [33].

Qualified³ is one of these recent approaches. They use unit tests to validate candidates code; they also include online interviews, a comparison tool between source code, and a detailed report of submission of the source code including time taken, test results, execution time, and output of the source code.

Another tool is Mettl⁴, which not only automatically qualifies source code, but also includes modules of psychometric and aptitude tests. Related to the source code, this tool retrieves a report including the numbers of attempts a coder tried to solve a problem, the time he/she spent and an overall performance of the coder.

Codility⁵ is another web platform built with the intention of fast filtering good candidates in a pool of applicants for a job. As they describe in their web page, during a hiring process, they noticed that the majority of the programmers they intend to hire were not able to program and they thought that the ones that were not able to solve a small problem may not been able to think in a complex distributed web application. In addition, they got a large amount of applicants in a small amount of time to decide, so they take inspiration in the Olympiad in informatics and built a web platform to automatically assess source code from the programmers they were intended to hire.

Another application is the one presented in [33]. They decode the human evaluation process extracting meaningful variables from it. They propose a rubric to grade students and also identify grammar features, data-dependency features, control context features, among others. Finally, they use simple feature selection and regression techniques, to build an evaluation system for testing computer programming skills.

When a programmer is applying to a job, it is indispensable to test his/her skills. Most of the time the test is made by giving a programming problem to the applicant and then measuring how good is her/his approach to solve the problems. Nevertheless, it is difficult to measure objectively how good a person is at programming if there is not a concrete rubric or a concrete number of features to measure within the source code. In addition, if there are hundreds of applicants, this could be exhausting and inefficient for the recruiter staff.

³<https://qualified.io>

⁴<https://mettl.com/>

⁵<https://codility.com/>

Giving an automatic assessment to a given source code to distinct good from bad approaches and its degree of correctness is not only useful in this professional context, but also could be very useful for instructors and students in the academic context.

3 A method for assessing source code using machine learning and source code analysis techniques

The process of grading source code assignments in a programming course, usually works as described in the Figure 3-1. The process could be divided in 3 steps: the first one (blue) starts when the instructor provides to the student the statement of a problem to be solved and the student programs a solution of the problem. The second one (green) happens inside an online judge, in this step, the student submits her proposed solution to the online judge for it to be tested against several test cases. This step finishes when the online judge answers her with a verdict, this verdict could be among the following:

- Accepted: All the test cases passed.
- Wrong Answer: At least one test case did not pass.
- Memory limit: The solution used more memory than the expected one at solving the problem.
- Time limit: The solution used more time than the expected one at solving the problem.
- Run Time Error: There was a run time error while running the solution.
- Compilation Error: There was a compilation error while executing the solution.

Finally, the third step (orange) is to provide the grade itself, in this case, the grade is binary depending on the verdict of the judge. The student will obtain a grade depending on the verdict on the online judge; she can only have a grade different than zero when the online judge provides the verdict accepted. Any other verdict from the source code is going to be mapped as a zero grade.

The aim of this work is to modify this process by adding a component before providing the grade to the student, i.e., a source code rubric and an automatic grader (the green dotted region at the bottom in Figure 3-1). The motivation is to provide a non-binary grade, but a

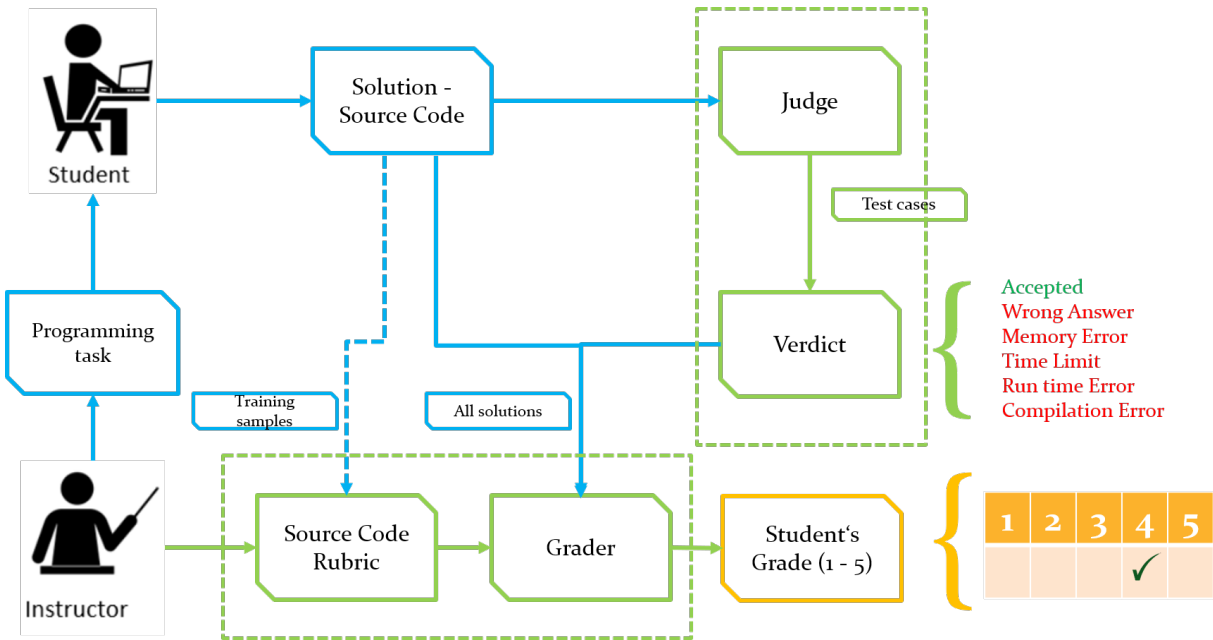


Figure 3-1: Proposed method

grade that has into account information extracted from the source code, from the verdict of the online judge, and from the test cases used. The proposed approach has two main steps: a rubric to grade source code and a grader.

The rubric was proposed to have an objective way to grade source code by having a check list that the source code should fulfill to have a certain grade. The idea is to use this rubric to grade some source codes made by the students to use these codes as samples of how a source code, with a given grade, should look like. Moreover, the grader is the component that receives as input source code graded with the rubric (to be used as training samples), source code we want to grade and information about it and the verdict. The grader will provide a grade between 1 and 5 to submissions we want to grade. The proposed step is shown in the Figure 3-1.

3.1. Source code rubric

The proposed rubric has 5 levels of grading, from grade 1 to grade 5, being 1 considered the lowest grade and 5 the highest one. The rubric is considering 4 main aspects in the source code: data structures used in the solution, test cases that each solution passed, how well the coder understood the problem, and the strategy used to solve the problem.

Those aspects try to be as general as possible for them to be used to measure the correctness of a wide amount of different problems. Also, because the rubric is taking into account several aspects of the source code, it tries to provide to the grader the elements to make a grade as objective as possible.

3.1.1. Building the rubric

The rubric was built in several stages starting with the 5 levels presented in Table 2-1 as first stage. The idea was to make the description of each level as clear and independent to each other to make the grading process as easy as possible for a human grader. To achieve this, the rubric was used to tag 35 wrong solutions from the dataset in groups of 7 problems each time. The problems were graded by 3 volunteers, with knowledge in algorithms and competitive programming. In every stage, it was asked to the volunteers, given the source code, the description of the problem and a correct solution, to try to classify each solution in one of the levels in the rubric, subsequently, we asked them to report how easy was to grade each solution. The reports were used to improve the description of each level in the rubric. In every stage the grades given to each problem by the volunteers where compared between them and the reports where shared to ask them if they had suggestions on how to improve the description of each level or if the description was not clear or was making the grading process difficult or blurry.

After 5 iterations, the volunteers manifested that to grade a solution was easy and that the levels were different enough to grade a problem among the levels of the rubric. Even though, it is worth to say that the volunteers manifested that the levels 3 and 4 were the most difficult to grade because the grader needs to understand fully the source code, and sometimes they faced problems as unknown data structures used, different and unusual ways to code and even the name of the variables name was a problem sometimes because they were confusing and in other languages.

3.1.2. Levels of the rubric

Level one

The first level tries to identify proposed solutions that could be classified, in general, as gibberish code. For example, proposed solutions showing no understanding of the problem, having data structures bad implemented and without logic, but also solutions that could be source code to solve another problem. This level is considered as the lowest grade in the rubric. In this level the source code does not pass any of the test cases proposed. Table 3-1 shows the details of the first level of the rubric.

Table 3-1: Description of the level one in the rubric

Characteristics	Level 1
Data structures	No logic
Test cases	It does not pass any test case
Understanding of the problem	It shows not understanding of the problem
Strategy used	It could be a strategy to solve another problem

Table 3-2: Description of the level two in the rubric

Characteristics	Level 2
Level before	It does not contain any characteristic belonging to the level 1
Data structures	It has the right data structures but they are not well implemented, bad used or they are not coherent
	It includes data structures different than the ones needed to solve the problem. It does not care how those data structures are implemented
Test cases	It does not pass any test case or passes a minimum amount of them (e.g., one or two)
Understanding of the problem	It shows a very basic or almost null understanding of the problem to be solved
Strategy used	The solution does not use a right approach to solve the problem. It could have used a brute force solution or another approach that does not work
	There is not a clear purpose in the source code and the used strategy (i.e., it is difficult to follow the logic of the source code)

Level two

The second level was made to identify solutions with emerging basic structures and a basic understanding of the problem. This level includes solutions that do not belong to the first level because they are, some how, trying to solve the problem, but they are not close to the correct proposed solution of the problem either. This level is also intended to identify solutions that are implementing other types of solutions beside the proposed as correct. For this case, it is understood that other solutions, beside the proposed one, can also lead to good verdicts from online judges, but we decided not to consider these solutions. For this level, source code should not have any of the characteristics from the level one. The Table 3-2 shows the second level of the rubric.

Table 3-3: Description of the level three in the rubric

Characteristics	Level 3
Level before	It does not contain any item from the level 2
Data structures	The right data structures needed to solve the problem exist and they are well implemented
Test cases	It could pass some test cases
Understanding of the problem	It shows a partial understanding of how to solve the problem
Strategy used	The strategy used is close to the one proposed

Level three

The level three of the rubric was made to identify solutions that are halfway to the correct solution. For this level, a solution should have the right data structures to solve the problem and they should be well implemented, having into account all the considerations to solve the problem. Moreover, the source code must show a partial but correct understanding of the problem. For this level, source code should not have any of the characteristics from the level two. Table **3-3** shows the characteristics of the level three.

Level four

The level four identifies solutions that are very close to the proposed correct solution. They show the right data structures and they are well implemented, additionally, the source code shows a complete understanding of the problem, and the solution is clear, easy to understand and in line with expectations but it is missing some small details, maybe one iteration more in a for cycle, a different data type than the expected ones, a wrong evaluation of a condition in a if statement or a problem with border cases. Table **3-4** shows the characteristics in the level four.

Level five

This level identifies the solutions that have right, well implemented, control structures, use the right amount of time to solve the problem and pass all the test cases. This kind of solutions has all the right data type, conditions, right amount of iterations and shows a fully and complete understood of the problem. Table **3-5** shows the description of the characteristic of the level five.

Table 3-4: Description of the level four in the rubric

Characteristics	Level 4
Level before	Not taken into account
Data structures	It has problems with the definition of the data types, conditionals or number of iterations
Test cases	It passes some test cases
Understanding of the problem	It shows a complete understanding of how to solve the problem
	It does not identify small details who lead to a not accepted solution(i.e., identify border cases)
Strategy used	The strategy used to solve the problem is clear, easy to read and according to the expected one

Table 3-5: Description of the level five in the rubric

Characteristics	Level 5
Level before	Not taken into account
Data structures	It does not have any problem with data types, conditionals, cycles, etc.
Test cases	It passes all test cases
Understanding of the problem	It shows a full understanding of how to solve the problem
Strategy used	The strategy used to solve the problem is clear, easy to read and according to the expected one

3.2. Grader

The grading phase has two main steps: the training and the classification. The training phase takes as inputs the source code graded by the instructor using the rubric meanwhile the classification task only takes the source code as the student made it and some information the online judge provides about it. This proposal is trying to emulate how a teacher reviews several solutions of a given problem, trying to understand how the students are solving the proposed task and later on grade several other proposed solutions.

The inputs of the grader are two: the proposed solutions made by the students to solve the programming task, some of them graded using the rubric for the training phase and some of them not graded for the classification task and information related to the programming task coming from the online judge.

The proposed solutions graded using the rubric are used for the training phase of the model and they are labeled by hand by the instructor, using her knowledge on how to solve the problem. She takes into account the required characteristics to belong to each level of the rubric as described in the Section 3.1.2.

The information extracted from the online judge is related to how many test cases each solution passed, the amount of memory and time used it used and the language it was written on.

Both of the phases, the training and the classification task require the data to be preprocessed and, right after, a phase of feature extraction, the Sections 3.2.1 and 3.2.2 describe those processes.

3.2.1. Preprocess of the data

After having the source codes, several processes were applied to them. The first one was to, using ANTLR¹, extract the Abstract Syntax Tree (AST) of every submission. This was accomplished using a well known C++14 grammar available in github². Another process applied was to tokenize each submission as well using ANTLR. An example of the tokens extracted for a piece of source code are shown in the Figure 3-2.

In addition, every submission is also operated as plain text, ignoring all the signs (such as plus, minus, equal signs, etc.) separating words in the source code by under scores or camel

¹A parser generator for reading, processing, structured text or binary files as described in <https://www.antlr.org/>

²<https://github.com/antlr/grammars-v4>

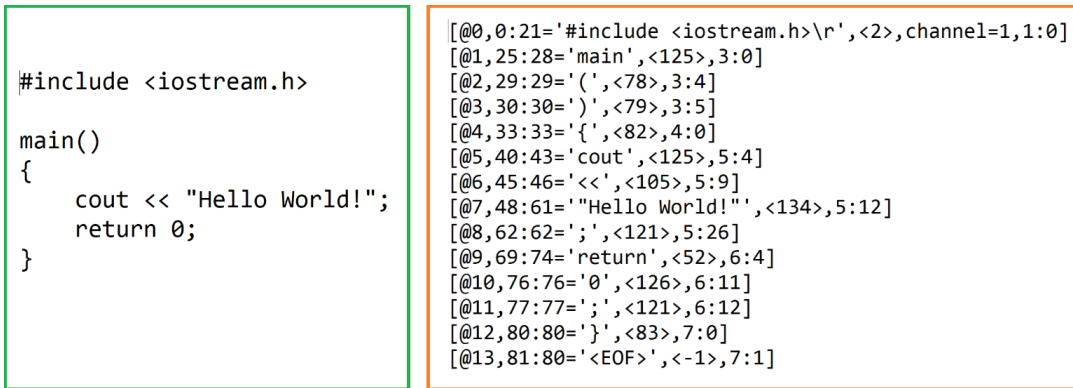


Figure 3-2: An example of a source code (green) and the tokens extracted using ANTLR (orange)

case given the case.

3.2.2. Feature extraction

Several representations were build from the source code pre-processing:

- A vector of 142 positions with the count of tokens in each submission. Figure 3-3 shows an example of a histogram of tokens of 451 source code examples solving a punctual task³. This Figure provides an idea of the tokens used to solve a task tagged as *dfs and similiar*. Table 3-6 shows the top 10 of the features that appear in that set of source code. The tokens are relevant because they give an idea of the variables and structures used in the source code.
- Features extracted from the AST as shown in Table 3-7. These features are aimed to extract characteristics from the syntax in the source code.
- N-grams from the AST as a string. This representation was build by creating the AST, transforming it to a string, and then, building n-grams from it as it was text. The most (orange) and the less (green) frequent n-grams are showed in the Figure 3-4.
- N-grams using the source code as text.

3.2.3. Training

Once the preprocessing and the feature extraction is done, we can represent every submission using the the features presented in the Section 3.2.2. For the training, it is required some

³<https://codeforces.com/problemset/problem/743/D>

Table 3-6: Top 10 of features appearance in a sample source code

Feature	Count
Integerliteral	169
Dot	51
LeftBracket	42
RightBracket	42
LeftBrace	41
RightBrace	41
Mutable	26
ArrowStar	20
Less	16
DotStar	15

Table 3-7: Source code features extracted from the AST

Definition	Count	Type
Number of if, -if -else, do, while, for, switch	6	Integer
Number of literals	1	Integer
Depth of the tree	1	Integer
Average of sons in the AST	1	Decimal
Number of nodes	1	Integer
Number if + Number of cycles + 1	1	Integer

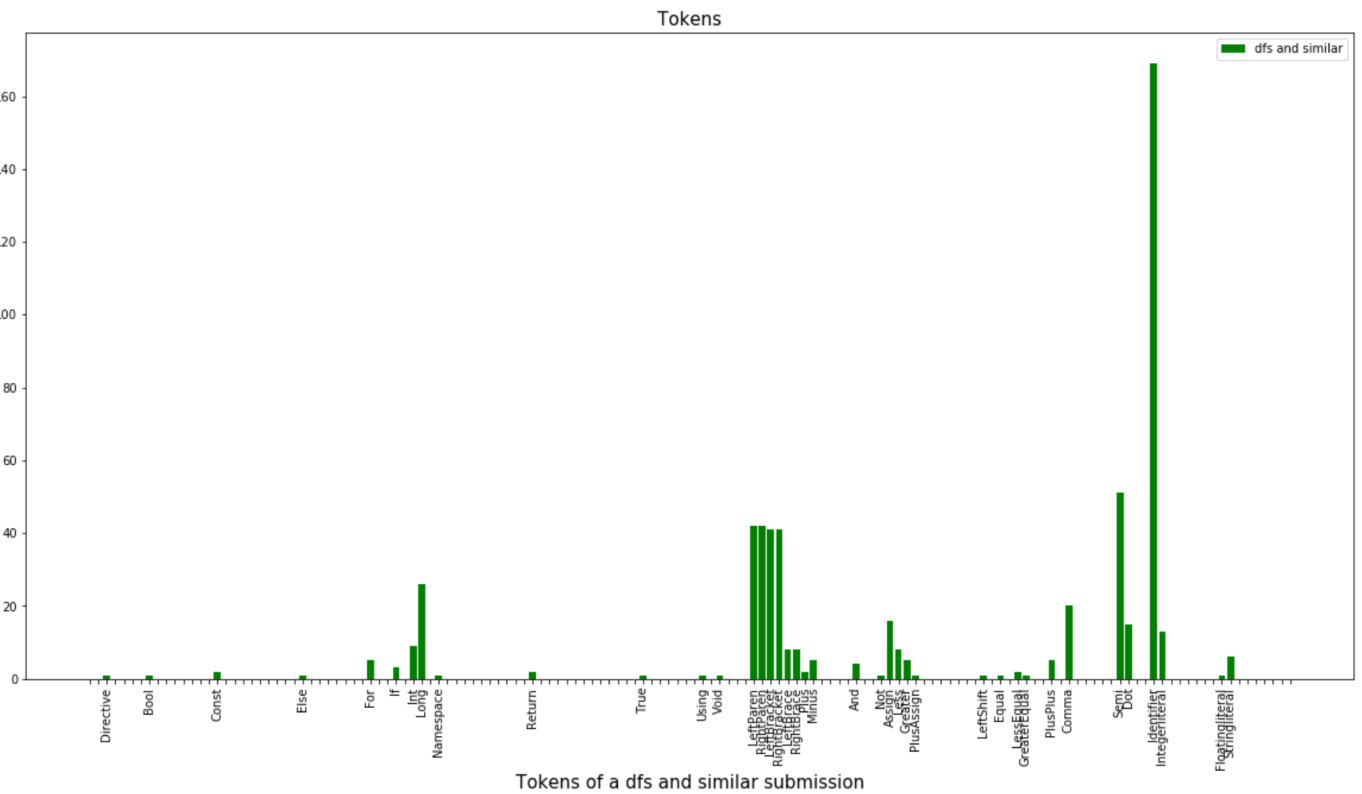


Figure 3-3: Histogram of tokens appearance for 451 samples of source code examples solving a punctual task

```
[(u'lint ) ', 1),
(u'"/proc/self/status"', 1),
(u'point {', 1),
(u'show ( result ) ;', 1),
(u'statement printf', 1),
(u'last , - 1 ,', 1),
(u'col = 0', 1),
(u'( cnt * ( cnt', 1),
(u' ) ; vector', 1),
(u'struct point', 1),
(u'( unaryoperator +', 1),
(u'int x ;', 1),
(u' ll ) ) )', 1),
(u'tcf ) ) )', 1),
(u'unqualifiedid Pll )', 1),
(u'( unqualifiedid vint64 )', 1),
(u'idexpression ( unqualifiedid sieve )', 1),
(u'NO" ; else cout', 1),
(u' || n', 1),
(u' ; if ( kaka', 1)]
```

```
[(u'( castexpression', 78417),
(u'( idexpression ( unqualifiedid', 78581),
(u'idexpression ( unqualifiedid', 78581),
(u'( idexpression (', 78871),
(u'( idexpression', 78871),
(u'idexpression', 78871),
(u'idexpression (', 78871),
(u'( unqualifiedid', 78872),
(u'unqualifiedid', 78872),
(u' ) (', 80893),
(u'postfixexpression', 104163),
(u'postfixexpression (', 104163),
(u'( postfixexpression', 104163),
(u'( postfixexpression (', 104163),
(u' ) ) ) )', 1172193),
(u' ) ) )', 1311269),
(u' ) )', 1468025),
(u' )', 1629913),
(u'( ', 1811298),
(u')', 1811299)]
```

Figure 3-4: A list of the most frequent (green) n-grams and less frequent (orange) n-grams for a set of 451 samples of source code solving a punctual task

examples labeled with the rubric presented in the Section 3.1 for the model to learn how a solution from a given grade of the rubric should look like. This step tries to mimic the part of when the teacher understands what the students are doing to solve a problem.

3.2.4. Classification task

The results of the classification task are presented in the Section 4, even though the general process starts when the model, being trained, receives a non-grade source code, based on what it has learn, it can classify the given source code as belonging to a grade of the rubric.

4 Experimental evaluation

This section describes the experimental setup, including a description of the dataset, how it was tagged, a description of the classification models, and how the parameters exploration was held; also, it presents the experimental results, and finally, the discussion of the results.

4.1. Experimental setup

4.1.1. Dataset

This section describes Codeforces¹ platform and details of the dataset selection. According to the founder, Codeforces is a platform to organize, implement and discuss competitive programming problems based on web 2.0². Founded in 2010, accordingly their yearly report, since 2013 is one of the most used platforms in community³, reporting during 2017 more than 35,000 active users⁴.

In the academy, Codeforces has been used as problem and exercise source during programming courses⁵. Recently, it has been used for source code analysis tasks [34] because of its variety and availability, and it has an API available to access data about the users, submissions, problems, etc.

Codeforces main structure is as follows: every 3 or 4 days, there is a weekly contest. Every contest has around 5 or 6 problems, ordered by letters, being the first one the letter A and the last one the letter E or F. Also, they are ordered by difficulty, being the A the easiest one in the contest, and the last one the most difficult one. From the very beginning, Codeforces implemented a ranking for the users based on the amount of points the user has reached during the contest. Table 4-1 describes the user ranking. A new user start in the specialist ranking by default, and depending on how many points she wins or losses during each contest, she will move in the ranking.

¹<http://codeforces.com/>

²<http://codeforces.com/blog/entry/1>

³<http://codeforces.com/blog/entry/10148>

⁴<http://codeforces.com/blog/entry/56831>

⁵<https://www2.cs.duke.edu/courses/spring14/compsci309s/syllabus.php>

Table 4-1: User ranking in Codeforces

Ranking	Points
Newbie	0 to 1199
Pupil	1200 to 1399
Specialist	1400 to 1599
Expert	1600 to 1899
Candidate Master	1900 to 2199
Master	2200 to 2299
International Master	2300 to 2399
Grandmaster	2400 to 2599
International grandmaster	2600 to 2899
Legendary grandmaster	2900+

Usually, the contests have a duration of two hours and a half, and they are divided in two categories, division 1 and division 2. The division 2 is opened for any user, but the division 1 is only available for users from the *expert* ranking to the top categories. In consequence, the problems in the division 2 are easier than the ones in the division 1.

During the contest, when a user tries to solve a problem and submits her proposed solution, she will get one of following answers (verdicts) from the online judge⁶:

- Memory limit exceeded
- Time limit exceeded
- Runtime error
- Wrong answer
- Idleness limit exceeded
- Denial of judgement
- Compilation error
- OK

Codeforces implemented a community where the contest are proposed by the users. The problem setters are in charge of writing the problem statement, the test cases and a tutorial to help out other users, also to categorize the problem in one or many of the 36 different tags (types of problems). It is important to remark that the tags are not mutually exclusive and that the problems could be solved with other techniques different than the ones proposed

⁶<http://codeforces.com/blog/entry/4088>

Tag	Number of submissions
implementation	9334472
math	4674132
greedy	4241116
brute force	3603542
dp (dynamic programming)	2694351
sortings	2255193
constructive algorithms	2086065
data structures	1870757
strings	1657799
binary search	1616874
dfs (Depth-first search) and similar	1373451

Table 4-2: Number of submissions per tag in Codeforces

by the problem setter, but in general the tags correspond to the solution that should be implemented by the users to guarantee an OK verdict given the time and memory limits. The top 11 tags with the most amount of submissions are presented in Table 4-2, Figure 4-1 shows the distribution of problems per tag with increasing order.

The dataset was built with source code submitted to Codeforces, using the API⁷ provided by the platform and a web scrapper built in python. The submissions and problems were collected from data available until July 31st 2017. At that moment there were 739 contests, 36 tags and 20,856,931 submissions classified over the mentioned tags.

All the submissions were submitted by users during real contest or simulation of real contest. Every submission contains the source code exactly how it was submitted by the user, a submission ID that identifies every submission as unique, the number of test cases passed, the memory and time used by the submission, and the verdict given the judge in Codeforces to the submission.

The scrapping process of the meta data about every submissions, such as the tags, test cases passed, memory used, time used and user was made using Codeforces API available for everyone to use and scrap data and Python 2.7. The submissions themselves containing the source code submitted were scrapped using Python 2.7 and HTTP request to Codeforces servers.

The source codes extracted were solutions from several problems in the platform solving an

⁷<https://codeforces.com/api/help>

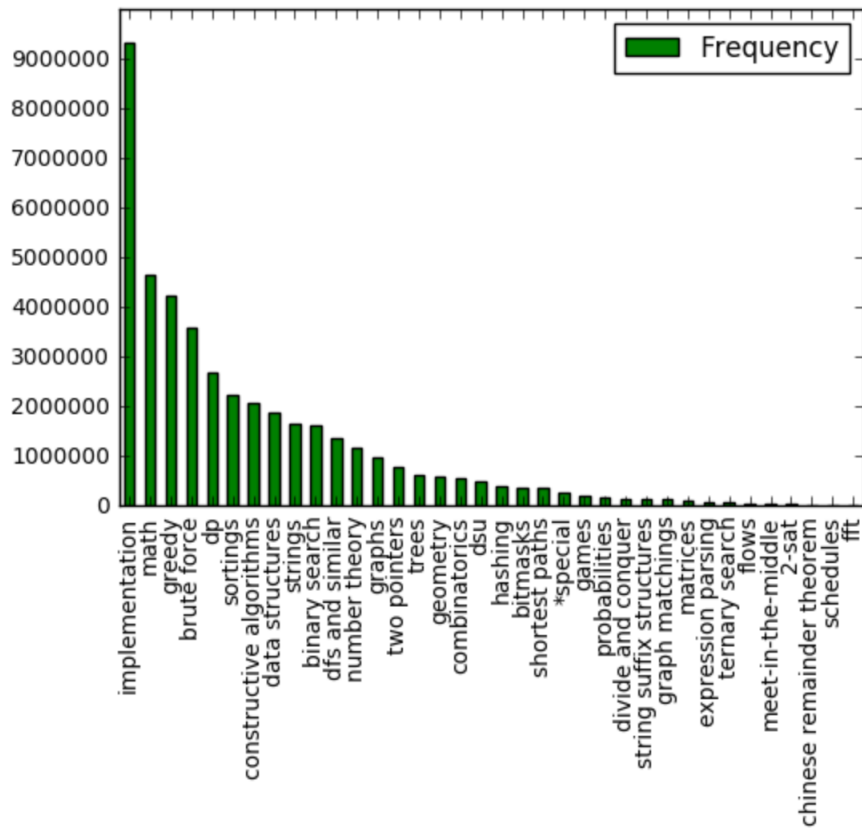


Figure 4-1: Distribution of the problems per tag

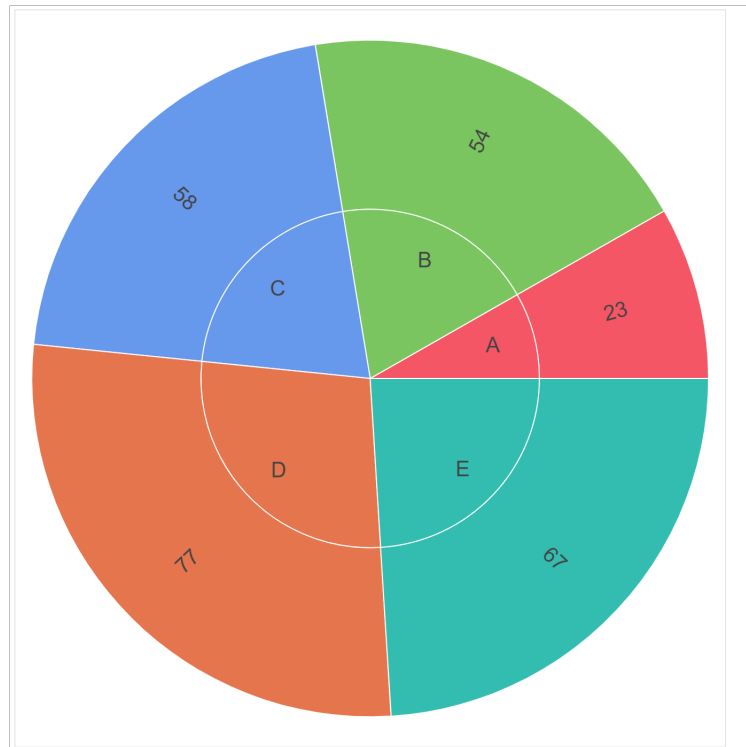


Figure 4-2: Distribution of the problems with the tag *dfs and similar*

specific type of problems tagged as *DFS and similar*. The tag *dfs and similar* was selected for the dataset because of number of submissions it has and also because the algorithms among this category, which are typically related to graph searches, have a well defined structure easy to differentiate among other algorithms.

The dataset contains 650 submissions scrapped from Codeforces.com ⁸, 550 belonging to grades 2, 3 and 4 from the rubric; and 100 belonging to the grade 1 of the rubric. As could be seen in the Figure 4-2, a big proportion of problems with the tag *dfs and similar* belongs to problems type B, C and D, which are problems of medium difficulty based on the experience of Codeforces contestants. In addition, as mentioned before, some problems could have more than one tag, it was considered the tags that appear with the *dfs and similar*. It is noticeable that the tags that appear with *dfs and similar* are tags related to graphs searches and type of graphs. Table 4-3 shows this relation.

As it was showed, Codeforces has several verdicts that could be considered as bad answers. Among those, the dataset was created using the followings:

- Memory limit exceeded

⁸<http://codeforces.com/blog/entry/1>

Table 4-3: Number of submissions with tag dfs and similar and tags related

Dfs Tag	Related Tags	Number of problems
dfs and similar	dp	142
dfs and similar	graphs	214
dfs and similar	trees	202
dfs and similar	dsu (disjoint set)	112

Table 4-4: Proportions of the verdicts per problems

	Wrong answer	Time limit	Memory error	Run time
745C*	0,906825	0,0605739	0,0495315	0,205882
764C	0,856973	0,714134	0,107095	0,338235
743D*	0,534421	0,0308183	0,0147256	0,0849673
750D	0,267656	0,443146	0,333333	0,124183

- Time limit exceeded
- Runtime error
- Wrong answer
- OK

During the construction of the rubric, it was noticed that the submissions with the verdict *Compilation error* could have content different than source code such as natural language or a combination between natural language and source code, a different language than the one specified by the user or any other content submitted that the compiler was not capable to recognize. Because of this, submissions with this verdict were not included in the dataset.

Among all the problems, 279 had the tag *dfs and similar*. Because the proportion of *wrong answers* verdicts was bigger, the amount of submission per verdicts was normalized by subtracting the maximum amount of submissions per verdict to each number of verdicts and dividing by the difference between the maximum amount and minimum amount of submissions with that verdict. This created values from 0 to 1 per verdicts among all the considered problems. After normalization, 2 problems were selected where the difference between the proportions per verdict were the smallest as possible. Beside this 2 problems, two other problems, 745C and 743D, were selected because they were used to define the rubric. Table 4-4 shows the proportions of the verdicts per problem.

The 556 submissions belonging to 2, 3 and 4 grades in the rubric were selected randomly from the problems showed in the Table 4-4. The other 100 belonging to the grade 1 where

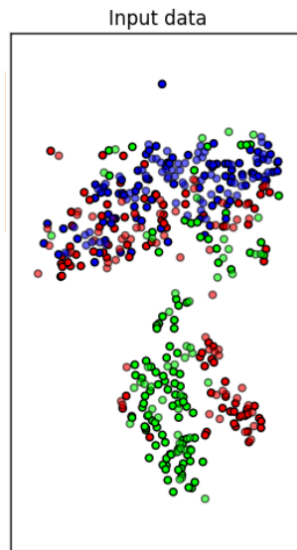


Figure 4-3: An early visualization of the submissions, in green the submissions with tag *math*, in blue the submissions with the tag *graphs* and in red the submissions belonging to the tag *dfs and similar*

took randomly from problems with the tag *math*. It was decided to select the tag *math* as the grade one because of early visualization experiments using a simple representation of source code such as token counts, showed that the problems with this tag were one of the most different ones to the tag *dfs and similar*. Figure 4-3 shows a visualization using a vector of tokens representations and PCA (Principal Component Analysis) as dimensionality reduction of 450 submissions, in red 150 belonging to the tag *dfs and similar*, in blue 150 belonging to the tag *graphs* and in green 150 belonging to the tag *math*. It can be seen that the submissions belonging to the tag *dfs and similar* and *graphs* mix together, meanwhile the submissions belonging to the tag *math* are far away from them and grouped together.

Coming up next, the problems with the tag *dfs and similar* were checked manually to guarantee they fulfill the following criteria:

- Assure they have a suggested solution implementing a graph search like a DFS (Depth-first search), BFS(Breadth-first search)or similar.
- Not a too complex solution to make the tagging with the rubric an easier task.
- To have a tutorial or an explanation of the problem in a blog in Codeforces and where possible, have the code proposed by the author available.

The distribution among the verdicts and problems is showed in the Figure 4-4.

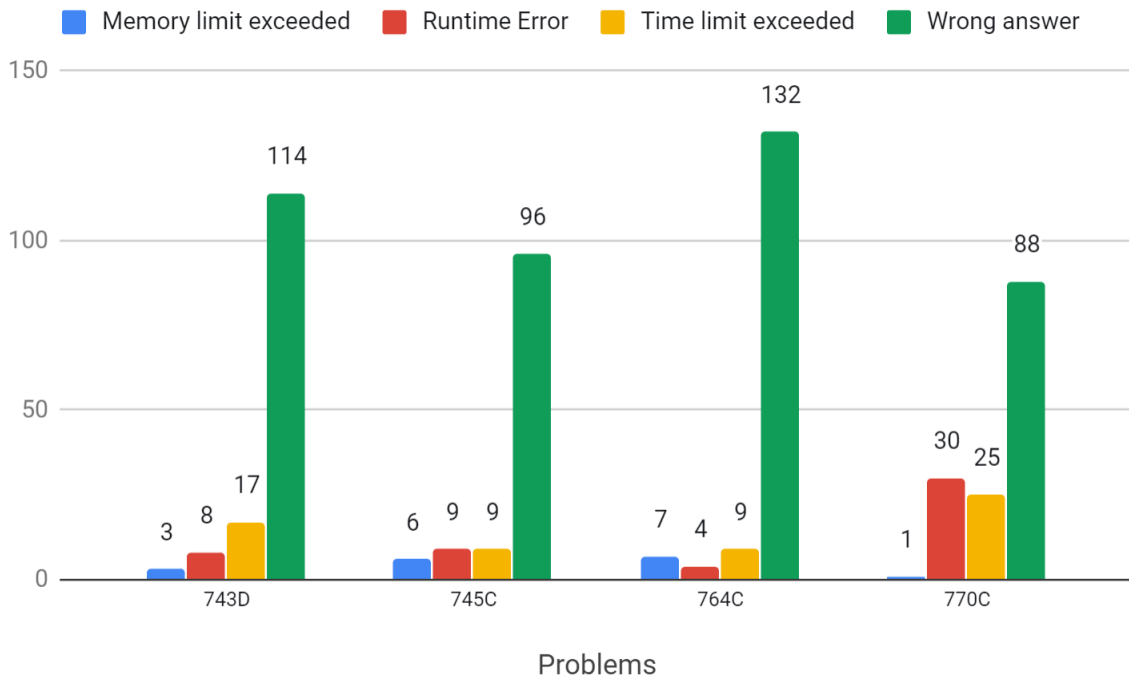


Figure 4-4: Distribution among the verdicts and problems

4.1.2. Tagging the dataset

The dataset was tagged using the help of a module from a system for learning programming skills called UNCode [27]. It is based on the grader INGenious [10, 11]. This grader was developed for both UNCode and the current work. It has a manual grader from where a rubric could be displayed, as can be seen in the Figure 4-5, it also shows the source code and information related to it such as the number of test cases passed, memory and time used by the solution. This module provides an easy and straight forward way to manually grade source code.

Having all the solutions in the UNCode grader, grouped by the tag in Codeforces, each solution was checked by hand using the module in UNCode, by two volunteers with knowledge in algorithms, data structures and the task to grade the solutions based on the rubric. This tagging process was made once per source code and at the end of the process each of the solutions have a grade from the rubric. Once the submissions were tagged, they were stored, as specified in UNCode in a MongoDB, from there the submissions were extracted to be used in the classification model. Figure 4-6 shows the graphic user interface in UNCode, which was used to label the dataset.

The tagged source code was classified among grades from 1 to 4 from the rubric. The sub-

Rubric scoring for Task: warmup

	Grade 1:	Grade 2:	Grade 3:	Grade 4:	Grade 5:
Level before	Does not apply.	Does not contain any item belonging to the level 1.	Does not contain any item from the level 2.	Does not apply	Does not apply.
Data structures	No logic.	Has the right data structures but they are not well implemented, bad used or they are not coherent.	The right structures need to solve the problem exist and are well implemented.	Has problems with the definition of the data types, conditionals or number of cycles.	Does not have any problem with data type, conditionals, cycles, etc.
Test cases	Does not pass any test case.	Does not pass any test case or passes one or two maximum.	Could pass some test cases.	Passes some test cases.	Passes all test cases.
Understanding of the problem	It shows not understanding of the problem.	It shows a very basic or almost null understanding of the problem to be solved.	Shows a partial understanding of how to solve the problem.	Shows a complete understanding of how to solve the problem Does not identify small details like border cases.	Shows a full understanding of how to solve the problem.
Strategy used	Could be a strategy to solve an other problem.	The solution does not use a right approach to solve the problem. Could have been using brute force or an other approach that does not work. There is not a clear purpose in the source code and in the strategy (i.e difficult to follow).	The strategy used to solve the problem is close to the one proposed.	The strategy used to solve the problem is clear, easy to follow and according to the expected.	The strategy used to solve the problem is clear, easy to read and according to the expected.

Figure 4-5: User interface in UNCode displaying the proposed rubric

missions with a grade 5 were selected from the dataset as ones with an *Accepted* verdict from the online judge in Codeforces. Table 4-5 shows the distribution of the submissions per grade in the rubric.

4.1.3. Classification models

The classification was held using several classifiers with significant results, such as:

- Support Vector Machine Lineal (SVM L)
- Random Forest (RF)

Table 4-5: Distribution of the submissions per grade tagged in the data set.

Rubric Score	Number of submissions
Grade 1	100
Grade 2	276
Grade 3	217
Grade 4	63
Grade 5	100

Rubric scoring for Task: warmup

[Source Code](#)
[Rubric context](#)

Source code of the submission

Programación de computadores Grupo 5

```

1
2 import java.util.*;
3
4 public class Main {
5
6     static MyBST<Integer> arbol = new MyBST<>();
7
8     public static class MyBST<AnyType extends Comparable<? super AnyType>> {
9
10        private BinaryNode<AnyType> raiz;
11        private static final int ALLOWED_IMBALANCE = 1;
12        private static int contador;
13
14        private class BinaryNode<AnyType> {

```

[Linter](#)

General Information

Author(s)	estudiante
Task	warmup
Language	java8
Submitted on	2019-03-05 16:48:25.153000

Comments

Data structures well implemented but partial understanding of the problem

Information

Memory	not memory
Test Cases	no test cases
Verdict	verdict

Scoring

Grade 1
 Grade 2
 Grade 3
 Grade 4
 Grade 5

[Grade](#)

Figure 4-6: User interface in UNCode used to label the dataset

Table 4-6: Best value of the parameter C for every representation

Representation	C value	f1-Score
Concatenation	-2	0.56
AST + Count Tokens (CNT)	-6	0.58
AST Features	-4	0.54
Count Tokens	-2	0.49
AST as Text	-6	0.45

It is worth to say that, during classification models exploration, other models were tested as well like Decisions trees, MPLClassifiers and Naive bayes. None of them with significant results.

For each of each of the models, the parameters exploration was made using a 10-fold stratified per class. To evaluate the results we used the f1-score, precision and recall. The classification was also made using the representations presented in the Section 3.2.2.

Regarding the Support Vector Machine, the exploration was made over the parameter C using a combination of several representations. Figure 4-7 shows the exploration of the parameter C for each representation. Every dotted line in the graph, shows the results of an f1-score of each representation. The dots represent the mean value with a given value of C and the areas around the lines show the standard deviation. We can see that the AST as text is the worst representation, the standard deviation is high and the results in terms of f1-score are the worst ones. On the other side, the concatenation of features works pretty well, being the concatenation of all features and the concatenation of the features extracted from the AST, and the count of tokens (CNT) the representations with best results in terms of f1-score, and in terms of having the smallest standard deviation. Table 4-6 shows the best value of the parameter C for every representation.

Figure 4-8 shows the exploration of deepness of the tree and number of estimators for the RF model using a concatenation of all representations as inputs. In addition, Figure 4-9 shows the exploration of same parameters only using the representation extracted from the AST of the source code. Figure 4-10 uses the parameters extracted from source code as text such as n-grams.

The best number for estimators using the concatenation of all representations and AST features was 50, on the other hand, for the features coming from the source code as Text, the best number of estimators was 100. Regarding the depth of the tree, for the concatenation of all representations and text-based features, the best depth of the tree was 28 and for the features coming from the AST the best results were found with depth of the tree

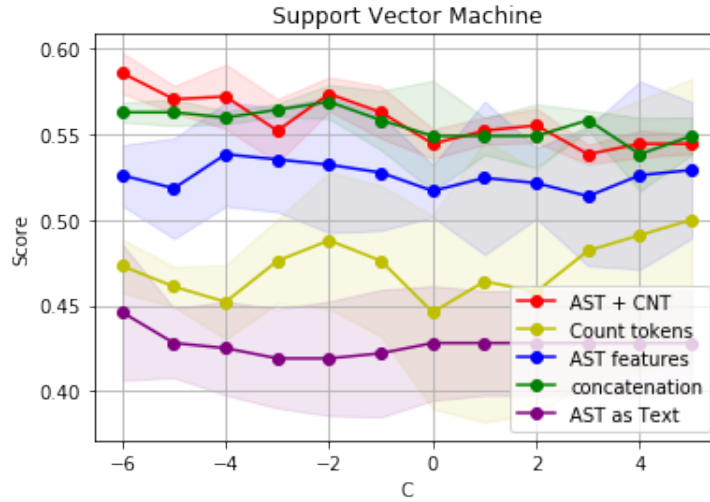


Figure 4-7: SVM L parameters exploration with several representations

Table 4-7: Recap of parameters exploration for RF

Representation	Parameter	Best Value
Concatenation of all representation	Number of estimators	50
	Depth of the tree	28
AST	Number of estimators	50
	Depth of the tree	10
Text (n-grams)	Number of estimators	100
	Depth of the tree	28

being 10. Table 4-7 shows the recap of the best parameters from the exploration with the RF model.

4.2. Experimental results

This Section shows the results of the classification, it will present the results per model and per feature. Table 4-8 shows the summary of the classification per different type of source code representation. As can be seen the best results were obtained using the concatenation of features, followed by tokens and n-grams of the source code as text. The following sections describe in more detailed the results with each of the representations by model.

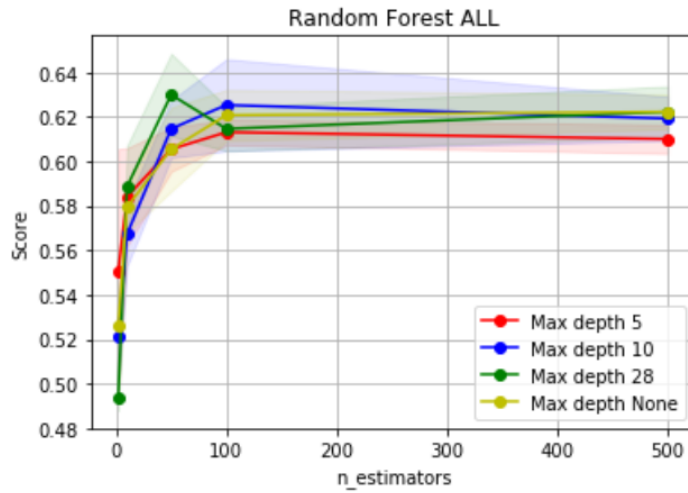


Figure 4-8: Random forest parameters exploration with concatenation of all representations.

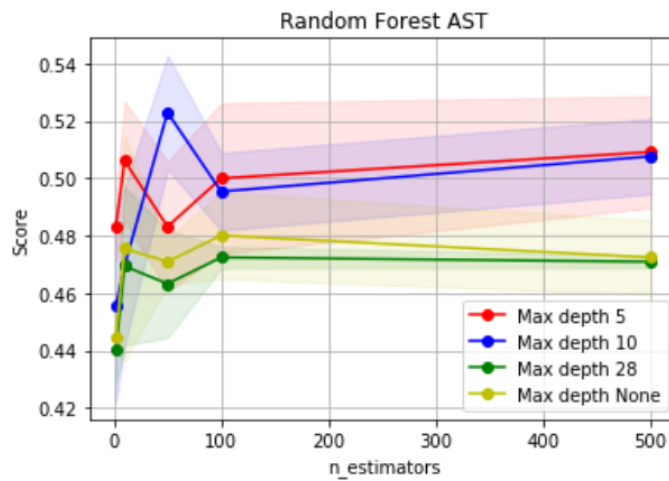


Figure 4-9: Random forest parameters exploration with features extracted from AST representation.

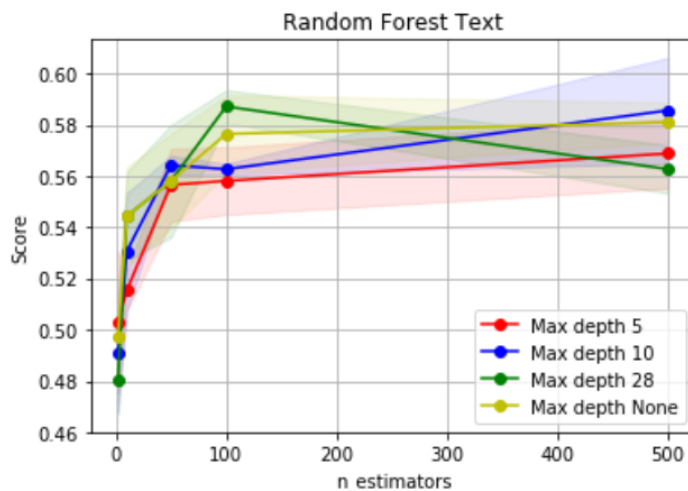


Figure 4-10: Random forest parameters exploration with features extracted from the source code as text.

Table 4-8: Summary of results with the classifiers SVM L and Random Forest and several source code representations.

Model	Representation	f1 Score	Precision	Recall
SVM L	Concatenation AST + n grams as text + tokens count	0.59	0.59	0.59
	Tokens count	0.56	0.56	0.59
	AST Features	0.52	0.59	0.54
	N-grams from AST	0.39		
	N-grams as text	0.37	0.50	0.49
Random Forest	Concatenation AST + n grams as text + tokens count	0.58	0.58	0.60
	Tokens count	0.59	0.58	0.61
	AST Features	0.51	0.51	0.53
	N-grams from AST	0.52		
	N-grams as text	0.54	0.55	0.56

4.2.1. Support Vector Machine Lineal (SVML)

Regarding the SVM L, Figures 4-11, 4-12, 4-13, and 4-14 present the confusion matrix for different source code representations. As seen in the figures, the confusion matrix in the Figure 4-13 suggest that the worse representation was n-grams from the source code as text, the model was classifying most of the submissions as grade 2 or grade 1, ignoring all the other categories. Almost all the submissions were classified as grade 2 and the f1-score was 0.37. By contrast, the best representation by it self was the tokens counting, which is a very simple and straight forward representation but it seems that for a lineal model it represents each grade more accurately than others representations. The best results were obtained concatenating all the representations.

The normalized confusion matrix in the Figure 4-11 shows the results of the classification using the concatenation of all features. It describes a *f1* score of 0.59, being the grade 1 in the rubric the easiest category to classify. This could be because these problems are the most different ones, in syntax, of the others because they are describing the solution of a entire different problem. The accuracy of the classification for this grade was 0.92, in contrast with the most difficult grade to classify that was the grade 4 only having an accuracy of 0.2. As seen, as the grade increases in the rubric, the more difficult it is to classify correctly the submission. Figure 4-12, shows the results classifying the submissions using features extracted from the AST, in this case, the representations works well classifying grades 1, 2 and 3 with accuracy of 0.8, 0.66 and 0.42 respectively. Figure 4-14 describes the results using the counting of tokens, which is the simplest representation used. In this case, likewise the results in the Figure 4-12, the normalized confusion matrix shows that the grades 1, 2 and 3 were easier to classify, but in this case, the grade 4 of the rubric had an small improvement.

It is worth to say that these results where expected as the grade 4 of the rubric was the most difficult to tag as expressed by the volunteers during the tagging phase. Even so, the volunteers expressed that it was difficult to differentiate from the grade 3 and the grade 4, but the results here show that the classifier is miss-classifying the submissions belonging to the grade 4 as grade 2 with most of the representations(features extracted from the AST, n-grams and tokens). On the other hand using, while using concatenation of features, the submissions where equally miss-classified as grade 3 or 4.

4.2.2. Random Forest (RF)

Regarding the Random Forest classifier, as seen in the Table 4-8, the precision, recall and F1 score were higher than when using the SVM L classifier. The only representation that did not outperform the SVML was the concatenation of AST, ngrams as text and tokens count. In general, as can be seen in the Figures 4-15, 4-17, 4-16 and 4-18, the easiest grade of

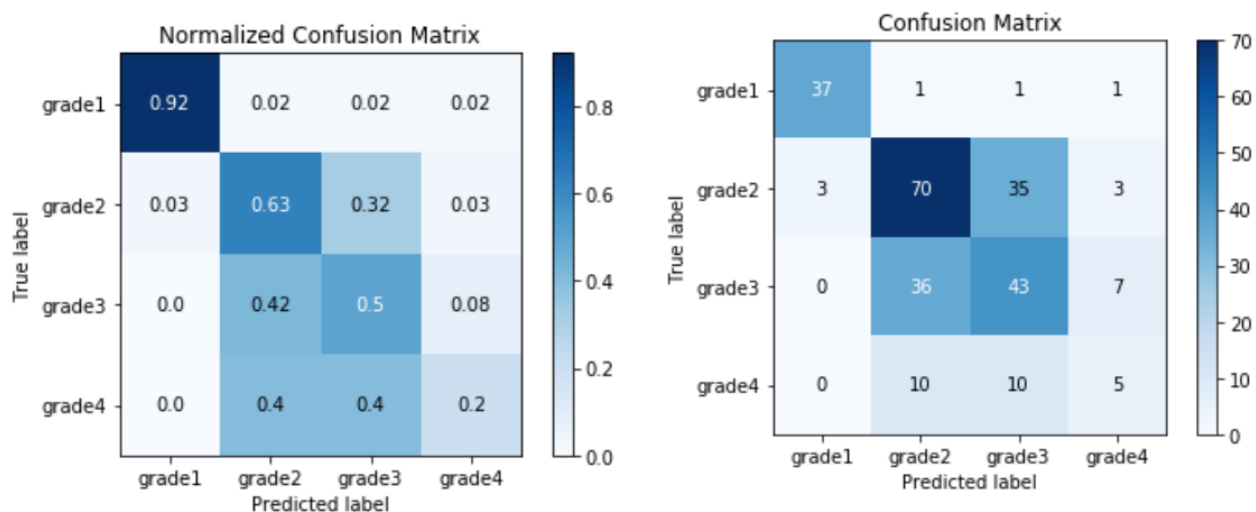


Figure 4-11: SVM L confusion Matrix from the classification using the concatenation of features extracted from the AST, n-grams as text and token counting.

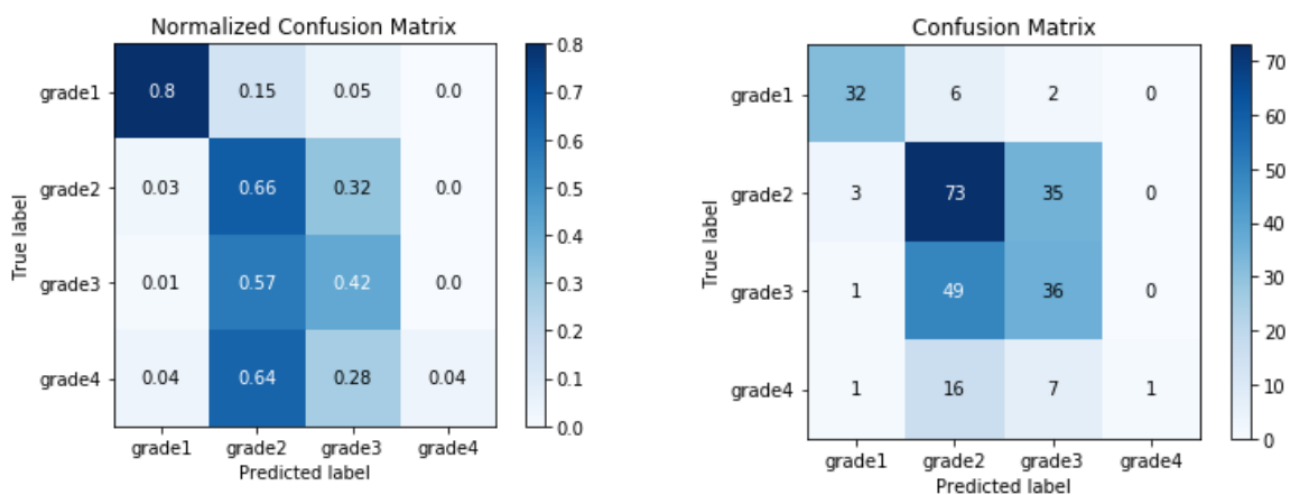


Figure 4-12: SVM L confusion Matrix from the classification using features extracted from the AST.

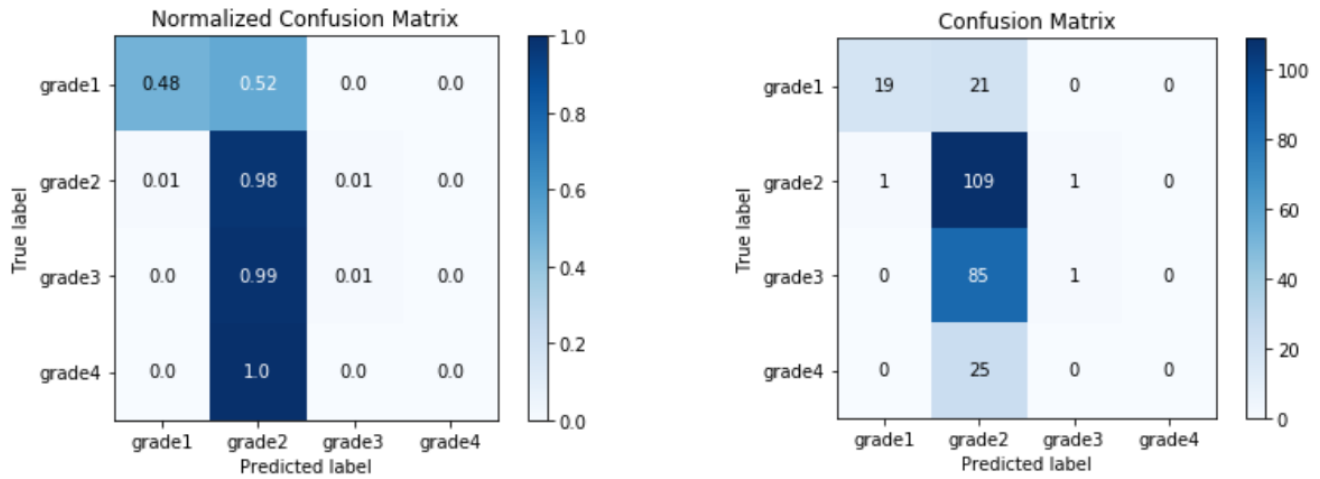


Figure 4-13: SVM L confusion Matrix from the classification using n-grams features using source code as text.

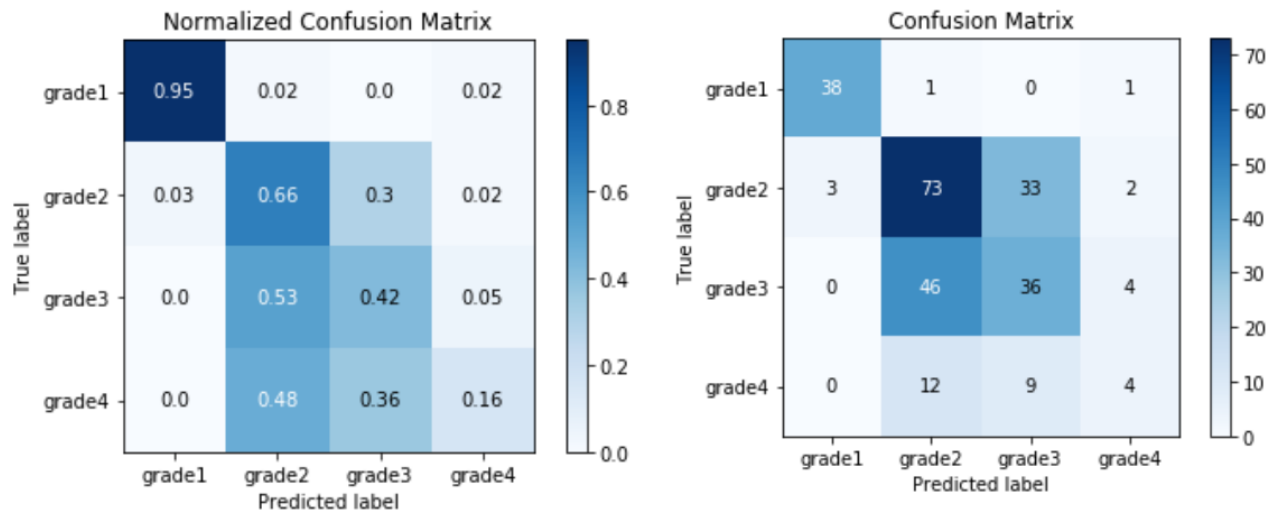


Figure 4-14: SVM L confusion Matrix from the classification using counting tokens as classification.

the rubric to classify was the grade one and the hardest one was the grade 4.

Figure 4-15 shows the results of the classification using the concatenation of all representations. For this case, the classifier did not perform well classifying the grade 4 of the rubric, was expected that it miss-classified most of the submissions with grade 4 as grade 3. Even though, the submissions originally belonging to grade 1 and 2 were very well classified, each of them with an accuracy of 0.85 and 0.74. Using these representations the classification of submissions belonging to the grade 3 were almost equally classified as 3 and 2 (0.48 and 0.49 respectively).

In the normalized confusion matrix in Figure 4-16, which describes the classification using the representation extracted from AST, the results are very similar to the ones described in the Figure 4-15 in terms of how the classifier behaves but in the confusion matrix described in 4-15 it has worse performance, miss classifying more submissions belonging to grade 1, 2 and 3. This representation had, overall all the representations the worse *f1* score (0.51).

Figure 4-17 describes the results of the classification using the ngrams extracted from the source code used as text. A remarkable characteristics in the normalized confusion matrix is that the accuracy per class was very similar and in this case the best results were obtained for the grade 2 of the rubric instead of the grade 1.

Finally, Figure 4-18 describes the results of the classification using one of the simplest representations: Token counting. In general, this representation had the best *f1* score with the RF classifier. The submissions belonging to the grade 1 and 2 were the most accurate accepted with 0.9 and 0.75 of accuracy each class. In addition, the submissions belonging to grade 4 originally were miss-classified mostly as grade 3, as said, which was expected. Even though, as the other representations, this one also had bad results to classify the grade 4 of the rubric.

4.2.3. Discussion

In general we can say that the representations of the source code matters a lot in this task and apparently simple representations can extract useful information to solve this task as tokens counts. This representation, being one of the simplest, had one of the best results in both classifiers. In addition, we can also see that when the more information we provide to the representations, for example concatenation features, we can improve the performance of the classifiers. Even though, we must be careful, because some features could also add noise, as can be seen in Table 4-8 where the concatenation of features had an slightly less *f1* score

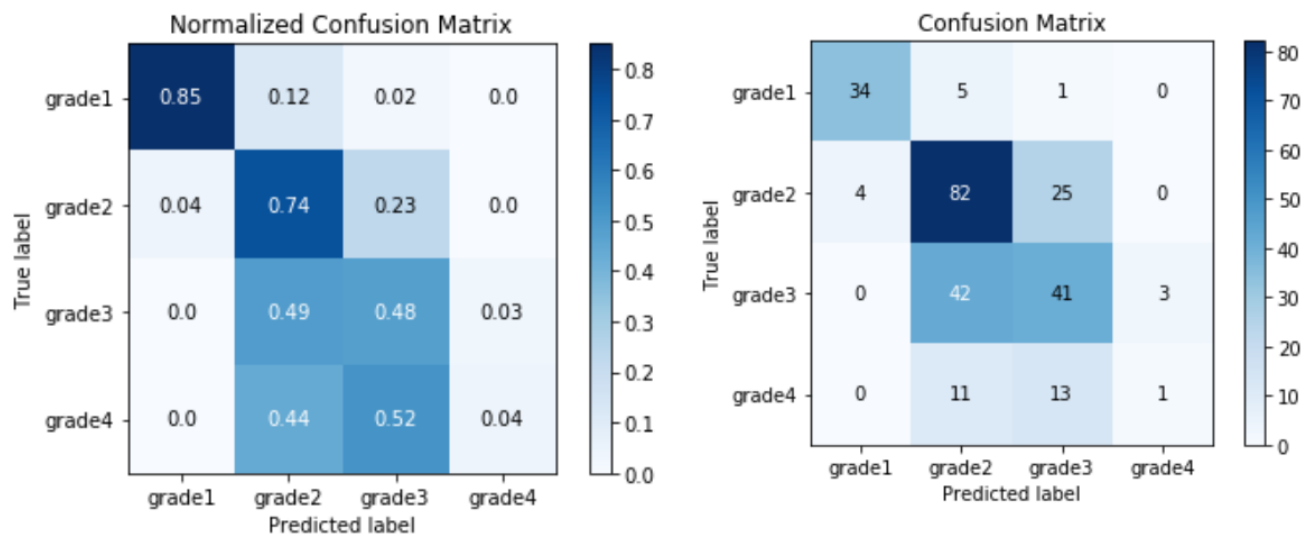


Figure 4-15: RF confusion Matrix from the classification using the concatenation of features extracted from the AST, n-grams as text and token counting.

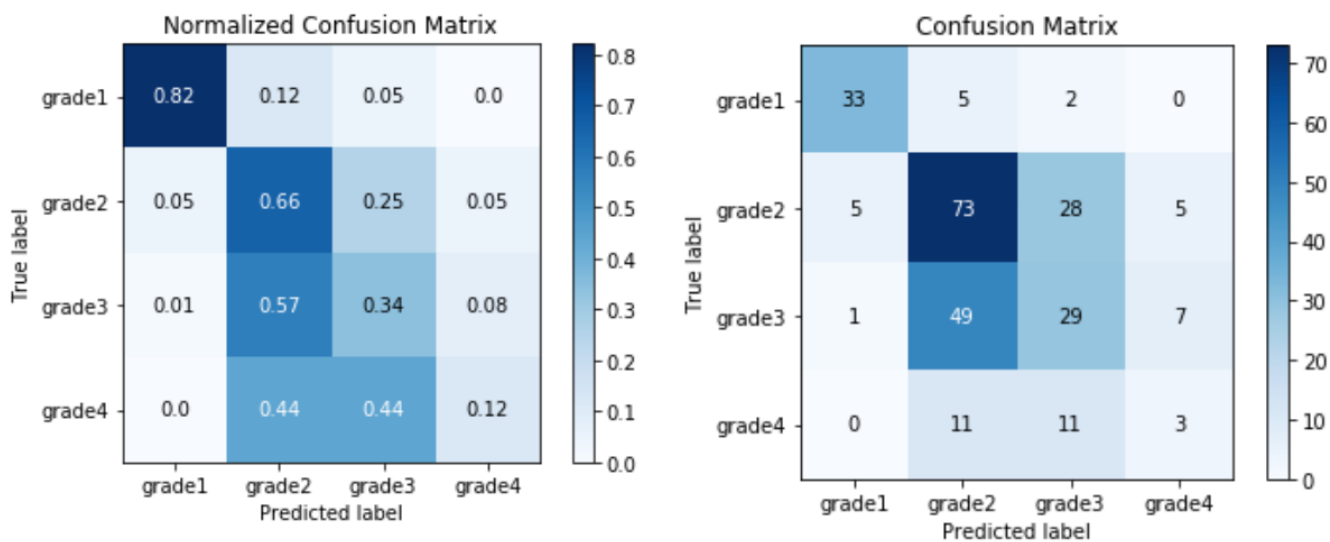


Figure 4-16: RF confusion Matrix from the classification using features extracted from the AST.

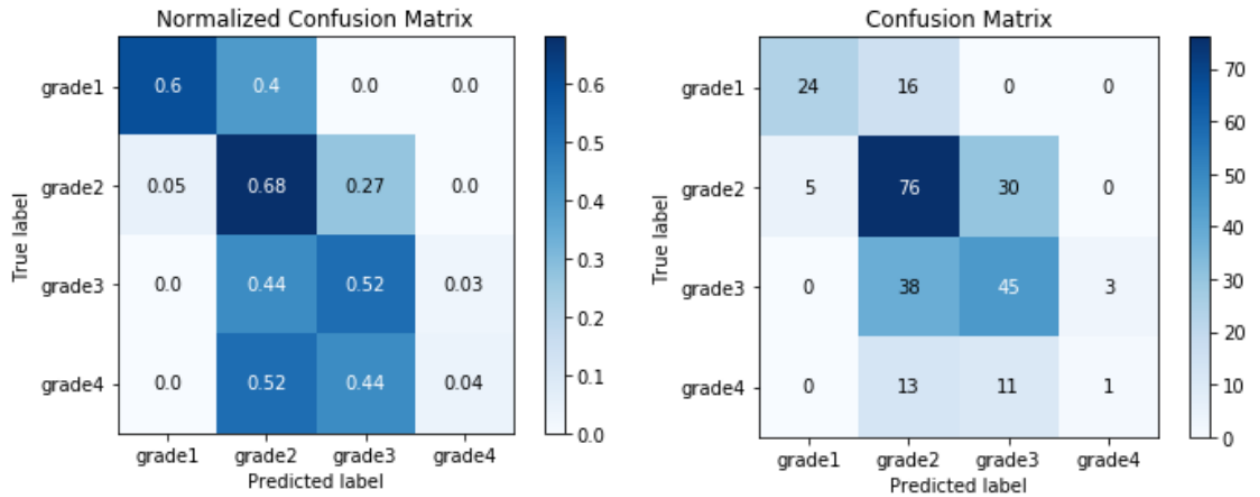


Figure 4-17: RF confusion Matrix from the classification using n-grams features using source code as text.

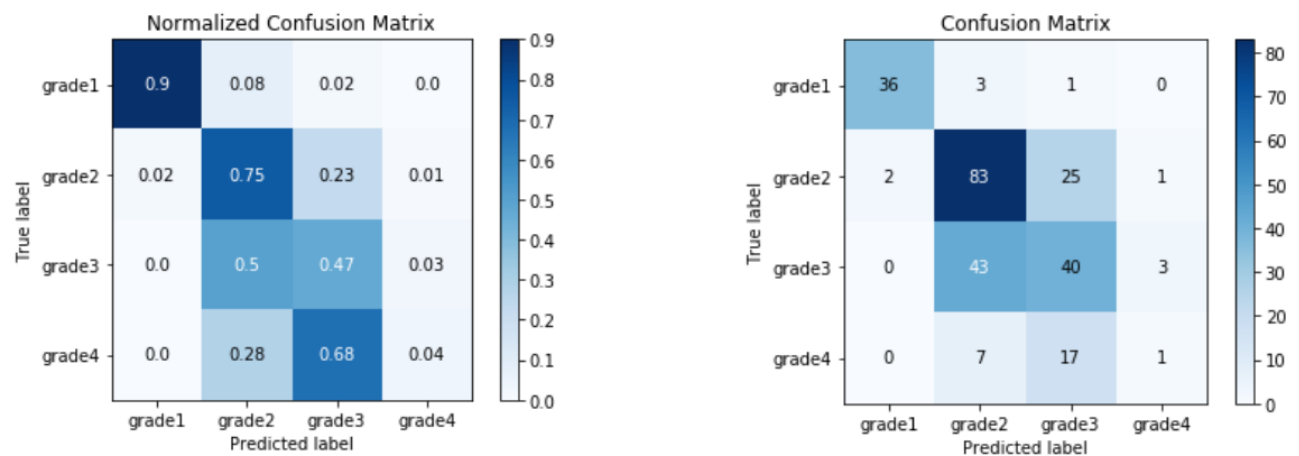


Figure 4-18: RF confusion Matrix from the classification using counting tokens as classification.

at solving the task.

Simple representations such as counting tokens are giving good results because they are identifying emerging basic structures such as counting expressions and variables declared, but also they are identifying control structures such as if-else statements, else, while, etc, and data structures like vectors, queues and stacks which are very meaningful for the *dfs and similar* kind of problems. Other representations, such as n-grams, try to identify the whole context of where every n-gram appears in the source code, this could be meaningful for the classifier, but as seen in Table 4-8 they do not perform very well by their own, especially when simple classifiers such as the SVM L are used. For both classifiers, concatenation of features and tokens counting had better results than the other representations.

The n-grams from the AST representation were expected to have better results, because it was going over the AST, which represents the syntax of the source code, and extracts the context of the components of the AST directly, even though the results were not as good as expected. This could be because the sense of depth of the tree was lost during the parsing because all the AST was treated as a long string. This representation and the n-grams from the source code as text had the worst *f1* score.

Related to the rubric, in section four we saw that there are some grades easier to classify than others. On the first hand, during the dataset-tagging phase, the volunteers expressed that while having the description in the rubric, identifying a source code that belongs to the grade one, two and three was very easy (specially the one and two). They expressed that the rubric was accurately describing each grade and the characteristics exposed where very easy to identify. Even though, they expressed that it was very difficult to identify submissions belonging to the grade 4 of the rubric because this grade was describing very small details that sometimes were hard to notice. We saw this behaviour reflected in the classification, for both classifiers, as can be seen in the Figure 4-11 and 4-15, the grade one and two were very well classified in the correct grade (*f1* score values among 0.74 to 0.92), the submissions belonging to the grade three were more or less well classified (*f1* score values among 0.48 and 0.50) and the grade four was poorly classified, with values of the *f1* score smaller than 0.2. It makes sense that the grade one of the rubric was very well classified because the submissions tagged with that label were not solving a *dfs and similar* kind of problem and the representations must have been reflected that. An example of this can be seen in the Figure 4-3.

5 Conclusions and Future Work

5.1. Conclusions

In this work we made several contributions to the automatic grading of source code. We made an exploration of how to visualize submissions of the students to a given task to help the teacher understand how the students are solving a problem [30]. We also proposed a method to include a model of automatic grading in a regular grading process, this model includes a rubric to grade source code, the exploration of several features to represent source code, the implementation of several machine learning classifiers to accomplish the grading task and the results of the classification task testing the model with a built data set.

A rubric is a very useful resource to help out an instructor to identify what grade a source code should have following a few well defined characteristics. This, by itself, is a significant help on trying to grade source code because it reduces the task of identifying possible problems during the resolution of a programming problem. Some benefits of this rubric is that it could be applied in general for different type or problems, not only the one used in the task we solved. It takes into account not only the data structures and understanding of the problem during solving the task but also the test cases, giving the chance to take into account a online judge if it is used and the strategy used to solve the problem.

The building of the data set graded with the rubric was a very time-consuming task, it implied obtaining the submissions by scrapping a well known online judge, filtering them and the revision of each submission by hand to decide whether if it belonged to a level of the rubric of an other. With the help of the module in UNCode we made this process easier, faster and user friendly. It is recommended to use data from courses where the instructor has a better understanding of the task that the students are solving or using any sort of unsupervised technique to filter out the submissions and help on the data set labeling. Another recommendations would be using crowd-sourcing web pages to do this tagging with the help of more volunteers all over the web.

Our approach graded with accuracy the submissions that belonged to the grade one and two and was having good results on identifying solutions that were half way of a correct solution (level three). The submissions that were very close to correct solutions were very difficult to identify, this was the worse level classified. Even though, based on our results, there is still a

long journey to actually do all the process automatically for all the levels of problems and in a general way, nevertheless providing a way for the instructors to save time identifying the worse implemented solutions comes very handy for a very common type of problems such as *dfs and similar*. It is worth to say that this task is very hard to solve in a general way, not only for the data set, but also for the wide amount of features that could be extracted from the source code, we presented the results of our model as a first approach on how to solve it. Even though, seen from the perspective of an instructor, the level one and two from the proposed rubric were describing the submissions that had the more different type of solution than the expected one, meanwhile the level three was identifying solutions that somehow were half way between a correct/expected solution. Identifying solutions belonging to the level one and two very accurately could help the instructor to filter out the submissions and focus of the problems that the students are actually having. On the other hand, identifying solutions that are halfway could be helpful because the instructor must provide a different type of feedback to these students because somehow, they are closer to a correct solution.

Talking about the source code analysis by itself, we explored different representations and features specially from the syntax and lexical aspects. We had very good *F1* scores with very simple representations such a tokens counting, on the other hand, representations that were more related to the syntax, extracted from the AST, such as identifying cycles, variables and control structures were not as accurate as expected. In addition, we approached the source code analysis also treating the source code as text, there we explored well known text representation such an N-grams. Nevertheless, was the concatenation of all representations the one with best results, this might imply that both, the syntax and the textual features provide useful information about the source code to solve this task, but the syntax plays a bigger role in the task.

To conclude, we provided a model, using machine learning techniques, that can be useful for instructors in terms on reducing time when grading source code by identifying accurately the students that are struggling the most on solving a task, this could reflect directly on the feedback that the students receive from the instructor. In addition we also provided insights on which source code features could be useful to solve this task.

5.2. Future Work

An immediate future task that could extend this work is to explore other ways to represent source code to try to extract more information from the submissions belonging to the level 4 of the rubric in order to get perform better at classifying this label. Also, it is necessary to try to other classification models different than SVM and RF. Other features that could be explored may include more syntax analysis over the source code.

Another task that could extend this work is to explore how to use the proposed rubric in other type of problems different than *dfs and similar* ones. This task may imply a complete different dataset, models and features. The submissions could not only belong to Codeforces, but also from other Online judges, coding interviews and programming courses.

To use the proposed model in another dataset, there might be a difficulty regarding to the dataset itself: the tagging phase was a very extenuating and long process because it has to be made by hand. A task that could have helped this procedure could be using a non supervised technique to filter out some submissions and make easier and faster the tagging process.

Bibliography

- [1] AGGARWAL, Varun ; SRIKANT, Shahank ; SHASHIDHAR, Vinay: Principles for using machine learning in the assessment of open response items: Programming assessment as a case study. En: *NIPS Workshop on Data Driven Education*, 2013
- [2] ALA-MUTKA, Kirsti M.: A survey of automated assessment approaches for programming assignments. En: *Computer science education* 15 (2005), Nr. 2, p. 83–102
- [3] ALSHAMSI, Fatima ; ELNAGAR, Ashraf: An automated assessment and reporting tool for introductory Java programs. En: *Innovations in Information Technology (IIT), 2011 International Conference on IEEE*, 2011, p. 324–329
- [4] AYHAN, Ülkü ; TÜRKYILMAZ, M U.: Key of Language Assessment: Rubrics and Rubric Design. En: *3rd International Conference on Foreign Language Teaching and Applied Linguistics* IBU Publishing, 2013
- [5] BOUDEWIJN, Nadia: *Automated Grading of Java Assignments*, Tesis de Grado, 2016
- [6] CALISKAN-ISLAM, Aylin ; HARANG, Richard ; LIU, Andrew ; NARAYANAN, Arvind ; VOSS, Clare ; YAMAGUCHI, Fabian ; GREENSTADT, Rachel: De-anonymizing programmers via code stylometry. En: *24th USENIX Security Symposium (USENIX Security), Washington, DC*, 2015
- [7] CARLESS, David ; SALTER, Diane ; YANG, Min ; LAM, Joy: Developing sustainable feedback practices. En: *Studies in higher Education* 36 (2011), Nr. 4, p. 395–407
- [8] COMBÉFIS, Sébastien ; DE SAINT-MARCQ, CLÉMENT [u. a.]: Teaching Programming and Algorithm Design with Pythia, a Web-Based Learning Platform. En: *Olympiads in Informatics* 6 (2012)
- [9] CSORNY, Lauren: Careers in the growing field of information technology services. (2013)
- [10] DERVAL, Guillaume ; GEGO, Anthony ; REINBOLD, Pierre. *INGInious [software]*. 2014
- [11] DERVAL, Guillaume ; GEGO, Anthony ; REINBOLD, Pierre ; FRANTZEN, Benjamin ; ROY, Peter V.: Automatic grading of programming exercises in a MOOC using the INGInious platform. En: *Proceedings of the 3rd edition of EMOOCs, the European MOOCs Stakeholders Summit*. Mons, Belgium, 2015, p. 86–91

-
- [12] FITZGERALD, Sue ; HANKS, Brian ; LISTER, Raymond ; MCCAULEY, Renee ; MURPHY, Laurie: What are we thinking when we grade programs? En: *Proceeding of the 44th ACM technical symposium on Computer science education* ACM, 2013, p. 471–476
- [13] GAUDENCIO, Matheus ; DANTAS, Ayla ; GUERRERO, Dalton D.: Can computers compare student code solutions as well as teachers? En: *Proceedings of the 45th ACM technical symposium on Computer science education* ACM, 2014, p. 21–26
- [14] GLASSMAN, Elena L. ; SCOTT, Jeremy ; SINGH, Rishabh ; GUO, Philip J. ; MILLER, Robert C.: OverCode: Visualizing variation in student solutions to programming problems at scale. En: *ACM Transactions on Computer-Human Interaction (TOCHI)* 22 (2015), Nr. 2, p. 7
- [15] GOSAIN, Anjana ; SHARMA, Ganga: Static analysis: A survey of techniques and tools. En: *Intelligent Computing and Applications*. Springer, 2015, p. 581–591
- [16] GRIFFIN, Merilee: What Is a Rubric?. En: *Assessment Update* 21 (2009), Nr. 6, p. 4–13
- [17] GUPTA, Surendra ; DUBEY, Shiv K.: Automatic assessment of programming assignment. En: *Computer Science & Engineering: An International Journal (CSEIJ)* 2 (2012), Nr. 1
- [18] HEXT, Jan B. ; WININGS, JW: An automatic grading scheme for simple programming exercises. En: *Communications of the ACM* 12 (1969), Nr. 5, p. 272–275
- [19] HSIAO, Chun-Hung ; CAFARELLA, Michael ; NARAYANASAMY, Satish: Using web corpus statistics for program analysis. En: *ACM SIGPLAN Notices* Vol. 49 ACM, 2014, p. 49–65
- [20] KATTA, Jitendra Yaraswi B.: *Machine Learning for Source-code Plagiarism Detection*, International Institute of Information Technology Hyderabad, Tesis de Grado, 2018
- [21] MIZOBUCHI, Yuji ; TAKAYAMA, Kuniharu: Two improvements to detect duplicates in Stack Overflow. En: *Software Analysis, Evolution and Reengineering (SANER), 2017 IEEE 24th International Conference on IEEE*, 2017, p. 563–564
- [22] MUSTAPHA, Aida ; SAMSUDIN, Noor A. ; ARBAIY, Nurieze ; MOHAMMED, Rozlini ; HAMID, Isredza R.: Generic Assessment Rubrics for Computer Programming Courses. En: *Turkish Online Journal of Educational Technology-TOJET* 15 (2016), Nr. 1, p. 53–68
- [23] NGUYEN, Andy ; PIECH, Christopher ; HUANG, Jonathan ; GUIBAS, Leonidas: Codewebs: scalable homework search for massive open online programming courses. En: *Proceedings of the 23rd international conference on World wide web* ACM, 2014, p. 491–502

-
- [24] PIETERSE, Vreda: Automated assessment of programming assignments. En: *Proceedings of the 3rd computer science education research conference on computer science education research* Open Universiteit, Heerlen, 2013, p. 45–56
- [25] PIETERSE, Vreda: Automated assessment of programming assignments. En: *Proceedings of the 3rd computer science education research conference on computer science education research* Open Universiteit, Heerlen, 2013, p. 45–56
- [26] PRECHELT, Lutz ; MALPOHL, Guido ; PHILIPPSEN, Michael: Finding plagiarisms among a set of programs with JPlag. En: *J. UCS* 8 (2002), Nr. 11, p. 1016
- [27] RESTREPO-CALLE, F. ; RAMÍREZ-ECHEVERRY, J.J. ; GONZALEZ, F.A.: UNCode: Interactive System for Learning and Automatic Evaluation of Computer Programming Skills. En: *EDULEARN18 Proceedings*, IATED, 2-4 July, 2018 2018 (10th International Conference on Education and New Learning Technologies). – ISBN 978–84–09–02709–5, p. 6888–6898
- [28] REVILLA, Miguel A. ; MANZOOR, Shahriar ; LIU, Rujia: Competitive learning in informatics: The UVa online judge experience. En: *Olympiads in Informatics* 2 (2008), p. 131–148
- [29] ROMLI, Rohaida ; SULAIMAN, Shahida ; ZAMLI, Kamal Z.: Automatic programming assessment and test data generation a review on its approaches. En: *Information Technology (ITSim), 2010 International Symposium in* Vol. 3 IEEE, 2010, p. 1186–1192
- [30] ROSALES-CASTRO, Lina F. ; CHAPARRO-GUTIÉRREZ, Laura A. ; CRUZ-SALINAS, Andrés F ; RESTREPO-CALLE, Felipe ; CAMARGO, Jorge ; GONZÁLEZ, Fabio A.: An Interactive Tool to Support Student Assessment in Programming Assignments. En: *Ibero-American Conference on Artificial Intelligence* Springer, 2016, p. 404–414
- [31] SCHLEIMER, Saul ; WILKERSON, Daniel S. ; AIKEN, Alex: Winnowing: local algorithms for document fingerprinting. En: *Proceedings of the 2003 ACM SIGMOD international conference on Management of data* ACM, 2003, p. 76–85
- [32] SRIKANT, Shashank ; AGGARWAL, Varun: A system to grade computer programming skills using machine learning. En: *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining* ACM, 2014, p. 1887–1896
- [33] SRIKANT, Shashank ; AGGARWAL, Varun: A system to grade computer programming skills using machine learning. En: *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining* ACM, 2014, p. 1887–1896
- [34] ZHANG, Hanrui ; CHENG, Yu ; CONITZER, Vincent: Quantitative Judgment Aggregation for Evaluating Contestants. (2019)