POLITECNICO DI TORINO
Department of Control and Computer Engineering
Master's degree in Computer Engineering

Master Degree Thesis

Evaluation of Students' Source Code Submissions Using Machine Learning

Supervisors
prof. Silvia Chiusano
prof. Siegfried Nijssen

Candidate
Simone Brigante

October 2020

Summary

In recent years, machine learning is experiencing a new golden age with the implementation of such methods in a wide variety of fields of study. Among them, the use of such techniques for source code analysis is gaining ground in tasks such as code optimization, code suggestion and bug detection.

This project is intended to be an introductory study for the application of these techniques in academic and didactic fields. In the course of this research we tried to analyze the source code developed by the students of the computer science course at the Ecole Polytechnique de Louvain. In particular, we tried to predict the outcome of the final exam taken by a student starting from the source code of the exercises carried out during the semester. In addition, a further analysis was conducted to understand whether the source code alone is sufficient to estimate the quality of an exercise. Finally, a study was made on the interpretability of the results obtained using the binary classifiers created.

The analyses undertaken did not always lead to the desired results, but can be seen as a first step in a field of study that deserves more attention.

Contents

List of Tables 6

List of Figures 8

1 Introduction 9
1.1 Content Overview 10

2 State of the Art 12
2.1 Evaluation of Students 12
2.2 Machine Learning on Source Code 14
2.2.1 Code2Vec 15

3 Context of Analysis 17
3.1 LINFO1101 17
3.2 INGInious 18
3.3 Database 19
3.3.1 Submission Collection 19

4 Analysis on Exercise Quality 21
4.1 Datasets 21
4.1.1 Exercises Extraction 22
4.1.2 Exams Extraction 23
4.1.3 Exercises and Exams Discrepancy 24
4.1.4 Datasets Description 24
4.2 Model Description 25
4.2.1 Gradient Boosting Classifier from Scikit-Learn 26
4.3 Analysis of Results 27
4.3.1 Attempted 27
4.3.2 Grades 29
4.3.3 Grades + Attempts 30
4.3.4 All Features 32
4.4 Final Considerations 33

5 Preparation for Source Code Analysis 35
5.1 Exercise Selection 35
5.2 External Resources 36
5.2.1 Astminer 36
5.2.2 Code2Vec 40
5.3 Dataset Description 44
5.3.1 Data Extraction 44
5.3.2 Data Preprocessing 45
5.3.3 Data Split 46
5.4 Code2Vec Execution 47
5.5 Data Postprocessing 47

6 Analysis on Exercise Source Code 50
6.1 Type of Models 50
6.1.1 Model on Code Vector 51
6.1.2 Model on Paths Vector 51
6.2 Exercises on Exams Outcomes 52
6.2.1 Dataset 52
6.2.2 Analysis on Code Vector 53
6.2.3 Analysis on Path Vector 54
6.3 Exercises on Exam Questions Outcomes 55
6.3.1 Dataset 55
6.3.2 Analysis on Code Vector 56
6.3.3 Analysis on Path Vector 59
6.4 Exam Questions Outcome 61
6.4.1 Dataset 61
6.4.2 Analysis on Code Vector 62
6.4.3 Analysis on Path Vector 65
6.5 Final Considerations 68

7 Interpretability of Code2Vec Results 70
7.1 Procedure Description 70
7.1.1 Attention Vector 71
7.1.2 Paths Extraction 71
7.2 Example 73
7.3 Considerations 75

8 Conclusion 76
8.1 Limits of the Analysis and Possible Improvements 77
8.1.1 Students Evaluation 77
8.1.2 Interpretability of source code 77
8.2 Application to Teaching 78

Bibliography 79

List of Tables

4.1 Averages of the summary variables 23
4.2 Exam sessions information 24
4.3 Datasets Discrepancy 24
4.4 Number of features in each dataset 25
4.5 Training and Test sets labels distribution for Exercise Quality Analysis 25
4.6 *Attempted* classification report 28
4.7 *Attempted* Features Importance sum after 1000 iterations 29
4.8 *Grades* classification report 30
4.9 *Grades* Features Importance sum after 1000 iterations 30
4.10 *Grades + Attempts* classification report 31
4.11 *Grades + Attempts* Features Importance sum after 1000 iterations . 32
4.12 *All Features* classification report 33
4.13 *All Features* Features Importance sum after 1000 iterations 33
5.1 Sample of entries in tokens.csv 38
5.2 Sample of entries in node_types.csv 39
5.3 Sample of entries in paths.csv + paths as string 39
5.4 Content of path_contexts.csv 40
5.5 Dimensions of datasets for source code analysis 45
6.1 Exercises on exam outcome dataset information 52
6.2 Exercises on exam outcome data split 53
6.3 Classification Report for *Exercises on Exam outcome* using model on *code vectors* 54
6.4 Classification Report for *Exercises on Exam outcome* using model on *path vectors* 55
6.5 Exercises on exam questions outcome datasets information 56
6.6 Classification Report for *Exercises on Exam Question q1* outcome using model on *code vectors* 57
6.7 Classification Report for *Exercises on Exam Question q2* outcome using model on *code vectors* 58
6.8 Classification Report for *Exercises on Exam Question q3* outcome using model on *code vectors* 58

6.9 Classification Report for *Exercises on Exam Question q6* outcome using model on *code vectors* 58
6.10 Classification Report for *Exercises on Exam Question q1* outcome using model on *path vectors* 60
6.11 Classification Report for *Exercises on Exam Question q2* outcome using model on *path vectors* 60
6.12 Classification Report for *Exercises on Exam Question q3* outcome using model on *path vectors* 61
6.13 Classification Report for *Exercises on Exam Question q6* outcome using model on *path vectors* 61
6.14 Exam Questions outcome datasets information 62
6.15 Classification Report for *Question q1* outcome using model on *code vectors* 64
6.16 Classification Report for *Question q2* outcome using model on *code vectors* 64
6.17 Classification Report for *Question q3* outcome using model on *code vectors* 64
6.18 Classification Report for *Question q6* outcome using model on *code vectors* 64
6.19 Classification Report for *Question q1* outcome using model on *path vectors* 66
6.20 Classification Report for *Question q2* outcome using model on *path vectors* 67
6.21 Classification Report for *Question q3* outcome using model on *path vectors* 67
6.22 Classification Report for *Question q6* outcome using model on *path vectors* 68
7.1 Most frequent paths for question *0119_q1* with number of appearances in each class prediction 72
7.2 Paths with higher attention from student submission 74

List of Figures

3.1 Example of an INGInious Exercise 18
4.1 *Attempted* Confusion Matrix 28
4.2 *Grades* Confusion Matrix 29
4.3 *Grades + Attempts* Confusion Matrix 31
4.4 *All Features* Confusion Matrix 32
5.1 *findMax* AST generated by astminer 37
5.2 *findMax* AST with labels 38
5.3 *Code2vec* model architecture 43
5.4 Dataset split 46
5.5 Steps of source code conversion 47
5.6 Pipeline steps 49
6.1 Confusion Matrix for *Exercises on exam outcome* using model on *code vectors* 53
6.2 Confusion Matrix for *Exercises on exam outcome* using model on *path vectors* 54
6.3 Confusion Matrices for *Exercises on Exam Questions outcome* using model on *code vectors* 57
6.4 Confusion Matrices for *Exercises on Exam Questions outcome* using model on *path vectors* 59
6.5 Confusion Matrices for *Questions on Questions outcome* using model on *code vectors* 63
6.6 Confusion Matrices for *Questions on Questions outcome* using model on *path vectors* 66
7.1 AST generated from the student's source code 74
7.2 Detail from the AST generated from the student's source code . . . 75

Chapter 1

Introduction

Research on the subject of machine learning has been growing more and more in recent years, especially due to the constant growth of computing power and the collection of massive amounts of data.

One of the areas in which machine learning techniques are being increasingly applied is source code analysis. Writing code is a job that requires great attention to detail and excellent problem solving skills. The application of machine learning in this field has many possible uses: from automatic code completion to code clones detection, but also techniques for code summarization or the ability to find small bugs.

The goal of this work is to see if it is possible to apply machine learning techniques to analyze source code files written by students of the computer science course at *Ecole polytechnique de Louvain* during the academic year 2018-2019. The aim is to see if it is possible to get predictions about their performance at the exam. Subsequently, the answers to the exam questions were analyzed to get information about their quality.

Making these assessments could improve the quality of the teaching by highlighting any recurrent errors made by the students during the course.

The main problem of applying machine learning algorithms for this kind of task is the fact that these algorithms need numerical representations as input, while the source code can be seen as a text file.

In order to solve the "conversion" problem we used *code2vec* a model designed by Alon et al. [2] capable of representing a snippet of code as a single vector of fixed length that can be used to predict the semantic characteristics of the code itself. It is able to do this starting from the representation of the code in the form of an *abstract syntax tree* (AST), to obtain which we used the program *astminer*.

To carry out our research, a pipeline has been developed that:

- extract the data necessary for our analysis
- manipulate them to make them fit our purposes

- run *code2vec* for source code conversion
- postprocess the data generated by the model

In the first phase of our analysis we tried to predict whether or not a student will pass the exam based on qualitative information about the exercises carried out during the year (such as the grade taken or the number of attempts made).

In the second part we focused on the study of the source code and in particular we tried to:

- predict the result of a student's examination by looking at the source code of the exercises uploaded during the course
- predict the outcome of a specific question of the exam starting from the exercises developed throughout the semester
- evaluate whether or not the raw source code is enough to estimate the quality of an exam question

All the analyses described have been carried out by means of the creation of two binary classifiers, using the *Gradient Boosting* model, one trained on the *code vectors* generated by *code2vec* and the other trained on more interpretable *paths vectors*.

The results obtained, although in some cases they did not meet expectations, are a good first step in the right direction.

One of the main limitations of this analysis was the relatively small size of the exercise datasets. In fact *code2vec*, but in general all machine learning algorithms, get better results with large amounts of data.

Among the possible future developments of the project, it would be extremely useful to aggregate the vectors of different exercises performed by the same student in order to have a much better overview of him. Moreover, in order to predict the outcome of an examination question starting from the exercises carried out during the year, it would be useful to select only a subset of them possibly related to the same subject of the question.

1.1 Content Overview

Chapter 2 contains a description of some articles about the two main themes of the project: the students' evaluation and the application of machine learning to the source code.

Chapter 3 gives a brief description of: the course of study analysed, the *INGInious* platform and the structure of the database.

Chapter 4 presents a first analysis in which, starting from the submissions saved in the *INGInious* database for each exercise, we try to predict whether or not a

student will pass the end-of-course exam. The features related to each exercise are: the best grade obtained, the number of attempts, after how much time since the beginning of the semester the student has done the exercise and the time elapsed between the first and the best attempt.

Chapter 5 describes in detail all the steps necessary for the source code analysis: the selection of the exercises, the data pre-processing phase, the execution of the model *code2vec* and the post-processing of the data generated by it. In addition, the functioning of the external tools used in our project *astminer* and *code2vec* is described in detail.

Chapter 6 presents the two types of models created for our analysis, one using *code vectors* and one using *path vectors*. Both are binary classifiers based on the *GradientBoostingClassifier* from *scikit-learn*. It also reports the results obtained for each of the three analyses, namely the prediction of the result of the exam (and of each single question) from the source code of the exercises carried out during the semester and the prediction of the result of an exam question from the source code developed for it by the student.

Chapter 7 proposes an example of study for the interpretability of the results obtained in the last of the analyses. In particular, it shows how certain *paths* can positively or negatively affect the evaluation of the code.

Chapter 8 presents the conclusions drawn from our study, describing the limitations and problems faced during its development and introducing some of the possible improvements for each analysis.

Chapter 2

State of the Art

The main focus of this project is to be able to predict the performance of a student based on the exercises submitted during the year by paying particular attention on the actual source code submitted.

This chapter will describe some of the studies done on the two core aspects of this project:

- *Evaluation of Students* - some of the approaches adopted in order to assess the performance of students during the year
- *Machine Learning on Source Code* - the possible applications of applying machine learning algorithms to the source code

Many research have been done about both of the topic using different methods, here we will try to summarise some of the interesting one read during the development of this project.

2.1 Evaluation of Students

Being able to predict the performance of the students has become object of study for many research in the academic field. Indeed, it is an important parameter to evaluate the quality of didactics of a whole university, but also of a specific course. In this way, professors can have a direct feedback to understand if their teaching method is effective or if some specific arguments need to be revised.

The growth of e-learning platforms has permitted to gain large amount of educational data which has allowed to start applying machine learning and data mining algorithms in order to extract useful information.

Shahirei et al. [19], in their paper, collect different methodologies on how to predict the progress of students using data mining techniques in order to improve teaching in schools.

In their research they take into account the two main factors that condition the various studies: the attributes chosen to measure performance and the prediction method to use with them.

In the various studies on this topic, many different types of attributes are taken into consideration, some apparently very distant from each other, but all of them allow to perform different analyses and to discover even unexpected correlations. The main ones identified in their study are:

- *CGPA (Cumulative Grade Point Average)* - is one of the most widely used attributes as a study parameter because it has the greatest impact on a student's academic and working future.

- *Internal Assessment* - i.e. the results obtained in the various projects, quizzes and labs carried out during the academic year, as well as attendance at lectures and workshops

- *Demographic* - such as gender, age, family background, etc.

- *External Assessment* - for instance extra-curricular activities and social interactions with fellow students.

- *Psychometric Factors* - such as the interest in learning new concepts, the method of study and the time spent on it, but also the support from the family. Although this kind of data is particularly difficult to obtain precisely.

Finally they quote a series of papers reporting the sets of attributes used, the data mining methods chosen and the results obtained in each search. Obviously among the most used methods appear: Decision Trees, Neural Networks, Naive Bayes, Support Vector Machine, etc.

In another research, Ashenafi et al.[3] study if it is possible to predict the exam scores of some computer science students from the history of their course activity. In their university students of the computer science faculty use a semi-automated peer-assessment system. This platform allows each student to submit a question on a particular topic of the course, the questions will be selected by the professors and than randomly distributed between the students. So everyone will answer the question assigned and, moreover, rate its interestingness, relevance and difficulty. Finally, each answer is also evaluated by the student who submitted it, but also from other ones who, of course, have not given an answer to the specific question. All these metrics are then preprocessed in order to train multiple linear regression models.

During the creation of the dataset, however, they realized that: all the students whose data was used to create the models had passed the exam. This shows that the students who participated in the teaching activities, even minimally, were able

to pass the exam. Therefore, the models that they created only manage to predict the grade of past exams (i.e. with a score between 18 and 30 out of 30).

They have developed two types of linear regression models: one in which they try to derive exactly the grade obtained by the student and the other in which they divide the grades into 5 buckets. To evaluate the performance of the models, they used the Root Mean Squared Error (RMSE) value and compared it with the results obtained by various random grading methods, showing that their models perform better than random guessing.

2.2 Machine Learning on Source Code

In recent years machine learning is living a new golden age thanks to the constant growth of the computational power of computers and the great interest in this topic by technology companies, but also in apparently very distant sectors.

In the research on this subject, the attempt to apply machine learning algorithms to the source code is becoming more and more popular. In the last few years, hundreds of papers have been published on this topic [5], conferences and even competitions have been organised [7].

As has been said, source code analysis using machine learning algorithms would have several applications, here are some of them:

- Source Code Analysis and Language Modeling
- Embeddings in Software Engineering
- Program Translation
- Code Suggestion and Completion
- Program Repair and Bug Detection
- APIs and Code Mining
- Code Optimization
- Code Summarization
- Clone Detection

For the purposes of this project the most interesting theme seemed to be the one about "Source Code Analysis and Language Modeling".

In their research, Zhang et al. [24] seek a new approach to analyse source code using machine learning algorithms.

A focal point of their research is how to represent code in a way that preserves its syntactic and semantic information. In fact very often the source code is treated

as if it were a natural language text, representing a program as a token sequences or a bags of tokens. However, unlike natural language, the code is characterised by a greater amount of information contained in each word and a much more complex syntax.

A better way to represent the source code is by means of *abstract syntax tree* (AST). They allow to better represent the syntax structure of the code. Each node in the tree corresponds to a construct or a symbol of the source code. In this way, details such as punctuation and delimiters are not taken into account, but emphasis is placed on lexical information and syntax structures. Other studies have also shown that syntax knowledge gives better results than tokenisation techniques. By combining AST with machine learning algorithms, e.g. Recursive Neural Network [23], Tree-based CNN [15] and Tree-LSTM [21], both lexical and syntactical information can be tracked. All these approaches, however, are subject to the gradient vanishing problem for very large and deep trees.

Zhang et al. [24] introduce "AST-based Neural Network" (ASTNN) a way to represent code fragments that divide large ASTs into sets of smaller trees capable of creating statement vectors that encompass lexical and syntactical knowledge. Their approach is generic enough to be used in several areas that require code understanding, such as source code classification and code clone detection, with very promising results in both areas.

2.2.1 Code2Vec

Finally, the research on which much of this project is based is that of Alon et al. [2] and their model **code2vec**.

Just as the distributed word representation used by *word2vec* [14] has had important repercussions in harnessing the power of neural networks for natual language processing (NLP) problems, in the same way *code2vec* wants to be a first step in the same direction in the study of source code.

The one presented by Alon et al. is a model capable of representing a snippet of code as a single vector of fixed length that can be used to predict the semantic characteristics of the code.

These methods of representing an object by means of low-dimensional vectors are known as *embeddings*. These vectors are able to distribute the meaning of a code element among several of their components, so that similar objects are mapped into similar vectors.

In their model this is done by decomposing the code into a set of paths extracted from the abstract syntax tree, learning a way to represent them atomically and, at the same time, learning how to aggregate them together.

Thanks to these *code embeddings* it is therefore possible to apply a wide range of machine learning algorithms to address several of the above problems related to the study of the source code. In their case they decided to focus on *semantic*

labeling of code snippets. That is, giving a descriptive name to source code, which implies finding a match between the entire content of a method and a label.

Their model receives the code snippet and a corresponding label as input, which expresses the semantic property that we want the model to learn. During the learning phase, vectors representing the individual labels are also generated. In the method name prediction task, these vectors have amazing properties, similar to the famous *word2vec* example: *vec(king) - vec(man) + vec(woman) = vec(queen)*. For example, if we sum the vector of the label *equals* and the one for *toLowerCase* we get one very close to the vector of *equalsIgnoreCase*, and it is also able to catch analogies like "*receive* is to *send* as *download* is to: *upload*".

Since machine learning algorithms almost always require numerical vectors as input, the code vectors produced by this model unlock the application of these algorithms for many tasks such as code retrival, captioning, classification and tagging, but also a metric for measuring similarity for code clone detection.

The results obtained from this research are very promising and that's why it was decided to use this framework as the basis for our study of the source code produced by the students.

A more detailed explanation of how *code2vec* works will be provided in the section 5.2.2, where we will comment on the main features of this model and how it was used for this project.

Chapter 3

Context of Analysis

3.1 LINFO1101

The analysis made during this project has been done on the data collected from the course *LINFO1101 - Introduction à la programmation* of the academic year 2018-2019 [13] at EPL in Louvain-la-Neuve.

The aim of the course is to introduce students to programming with the *Python* language. They are taught how to analyse a problem in order to implement a solution and the main concepts of object-oriented programming and simple data structures.

During the year each student has access to an online syllabus [20] which contains: the theoretical notions taught during the course and many exercises to test their ability and their knowledge on each topic.

The theory part is divided in three main topics:

1. *Introduction* - which gives the basic concepts of programming like: what is a program, the description of variables, expressions and statements, how to manage conditional execution and iteration and how to import and use modules.

2. *Data Structures* - which describes how to handle data in: strings, lists, tuples, dictionaries, etc. In addition there is an introduction to files, search algorithms, testing and exception.

3. *Objects* - where are explained the basic and advanced concepts of class and object such as polymorphism and inheritance.

The exercise part is divided, instead, in 12 missions each one about a specific notion faced in the theoretical lectures. There are different kind of exercises: multiple choice questions, completion of code snippets or uploading a whole Python script.

The exercises done by the students are the main object of analysis of this project and the process of selecting and evaluating them will be described further in detail later in the chapter 5.1.

3.2 INGInious

The exercises from the syllabus cited earlier are tested on INGInious [22], a platform developed by the INGI department at UCLouvain, used to automatically grade the programming exercises uploaded by the students.

The tool is written entirely in Python, it relies on Docker in order to provide a secure execution environment in which to run the programs. INGInious uses MongoDB as a database to keep track of submissions and store them together with other information such as the student identifier, the time of the submission and the python file itself.

Moreover, INGInious is completely language-agnostic and is able to run anything, even though it is currently limited to Linux programs given that only Linux containers are provided and supported.

Figure 3.1: Example of an INGInious Exercise

Once a student submit a programming exercise, INGInious automatically execute it on a container and runs some tests, specifically made for the task, in order to graduate the work of the student. The tests usually cover different inputs and corner cases so that the exercise can be better evaluated in a scale between 0 and 100% according to the weight of each test, in this way we have a larger range of grades to rate the performance of a student.

INGInious is used also for the final evaluation of the students. Indeed, all the students perform their exam on the platform, where each question is presented as a task with a detailed description, an area in which to write their solution (which usually consist in the implementation of a method) and a "testing" area where the student can run some custom test on his solution.

Finally, INGInious offers to course administrators and tutors some tools in order to have a better understanding of the students' performances. They can see statistics like: the number of submissions during a certain period of time for each task, the progress of each student during the semester and so on. They can even generate a report with some graphs showing the performances of the class on each task of the exam.

3.3 Database

All the exercises submitted by the students are stored in a MongoDB database. Here are stored all the collections need to run the application. The most important one for our analysis is the *submissions* collection which, as the name suggests, contains all the submissions from all the students of all the courses.

3.3.1 Submission Collection

Even though MondoDB is a relational database, so each document in a collection might have different fields, all the submissions present the same structure. Here there are the main fields of a *submissions* document used to perform our analysis:

- *_id* - unique identifier of the submission
- *courseid* - course identifier
- *taskid* - task identifier
- *username* - list of usernames of the students who have submitted the exercise (in order to handle group projects)
- *submitted_on* - date of the submission
- *input* - id of the actual file submitted retrievable by means of GridFS [11]
- *grade* - the evaluation of the exercise in a percentage format

The data is queried from the database by means of the libraries:

- *pymongo* - a Python distribution containing the tools for working with MongoDB[16]

- *gridfs* - a specification to store and retrieve files that exceed the BSON-document size limit. GridFS splits the file into smaller chunks and stores each of them as a separate document [11]

Chapter 4

Analysis on Exercise Quality

This chapter will describe the first analysis carried out on qualitative data extracted from the *submissions* collection. It will be divided into the following sections:

- *Datasets* - where will be described the process of extracting the exercises from the database and the creation of the features used.
- *Model Description* - in which we will explain how Gradient Boosting, the model chosen for our analysis, works.
- *Analysis of Results* - where the experimental results obtained will be shown.
- *Final Considerations* - in which the conclusions of our findings will be drawn.

The results obtained in this phase will be taken as a starting point for this project. The aim of this analysis is to find out if it is possible to predict whether or not the student will pass the exam using different sets of features.

4.1 Datasets

The generation of the dataset has been divided into two phases:

- the extraction of the results of the exercises carried out by the students in order to generate the features
- the extraction of final exam results in all three examination sessions in 2019

Once the necessary features were extracted, four datasets with an increasing number of features were created in order to evaluate the contribution they make to the accuracy of the models.

4.1.1 Exercises Extraction

Within the syllabus of the *LINFO-1101* course of the academic year 2018/19 there are 113 different exercises, some are simple multiple choice quizzes, others require you to run complex scripts in Python. For this analysis it was decided to use all the available exercises because the output of the evaluation is in the same format for all of them.

In order to build the features dataset we have extracted, from the *submissions* collection, for each exercise:

- the username of the student who did the exercise
- the grade automatically assigned by INGInious
- the date on which the student did the exercise

Then, for each exercise done by each student, the following parameters were extracted:

- *best_grade* - as the name suggests, it is the best grade obtained by the student for a given exercise.
- *attempts* - the number of times a student has done and uploaded the exercise.
- *time_to_best_grade* - the time, in hours, between the first and the best attempt for a given exercise.
- *days_since_beginning* - the time, in days, between the beginning of the semester and the time of the first submission.

The first two features are quite obvious, the other two are designed to try to understand if time variables can offer interesting hints to judge a student's school performance.

Time_to_best_grade has been introduced to assess whether the time spent by a student on a given exercise can be correlated with a better exam result. For example, a good student could do an exercise perfectly on the first attempt or, in any case, in a short time compared to a less attentive student.

Instead, *days_since_beginning* has been considered because all the exercises are available throughout the year, so it may be interesting to see if the period in which an exercise is done affects the student's performance. For example, you might be able to tell from the results if doing the exercises one at a time is more or less effective than doing them all close to the exam date.

Finally, since each student is represented by a vector containing all the features for each exercise, summary features have been added to provide a general overview for each student:

- *avg_best_grade_tried* - the average best grade computed only over the exercises attempted
- *avg_best_grade_total* - the average best grade computed over all 113 exercises
- *tot_100* - the number of exercises in which the student got a full score
- *tot_attempted* - the number of exercises attempted
- *avg_attempts* - the average number of attempts on all the exercises attempted
- *avg_time_to_best_grade* - the average time_to_best_grade over the exercise attempted
- *avg_days_since_beginning* - the average days_since_begin over the exercise attempted

A total of 741 students did at least one of the exercises proposed during the semester and the table 4.1 shows the averages of the summary variables:

Summary Feature	**Average**
avg_best_grade_tried	86.70
avg_best_grade_total	49.45
tot_100	55.29
tot_attempted	62.16
avg_attempts	5.93
avg_time_to_best_grade	119.37
avg_days_since_beginning	87.78

Table 4.1: Averages of the summary variables

4.1.2 Exams Extraction

A similar approach to that described above has been followed for the extraction of the exams data.

In INGInious each exam session is saved as a separate course (*LINFO-1101-0119*, *LINFO-1101-0619*, *LINFO-1101-0819*) so that it is easier to manage enrolments and obtain student performance reports. An exam consists of 6 questions each with its own weight, the maximum score for an exam is 20 and an exam is considered passed if the student reaches a score of 10 out of 20.

As in the case of the exercises, the best submission was taken for each question of the exam and the total score obtained was calculated for each student.

It was decided to consider all three exam sessions held during the school year to take as many students as possible.

In case a student participated in more than one examination session, for example if they failed to pass the exam on the first attempt, the most recent attempt was taken into account. In the table 4.2 we find a summary of the enrolments and the number of students who have passed the different exam sessions.

Session	Students Enrolled	First Attempt	Passed	Passed (%)
Jan 19	615	615	253	41.14
Jun 19	152	60	107	70.39
Aug 19	172	37	100	58.14
Total	/	712	460	64.61

Table 4.2: Exam sessions information

4.1.3 Exercises and Exams Discrepancy

Unfortunately, the number of students who have done the exercises during the semester does not match perfectly with those who have subsequently taken the exam.

In fact, there are 741 students who have done the exercises and 712 who have taken part in the exams, but only 466 students belong to both groups. This may be due to the fact that some students may have taken the course without attempting the exam or, on the contrary, taken the exam without having done any exercises, perhaps because they took the course the previous year.

Set	#
Students from Exercises	741
Students from Exam	712
Intersection	466

Table 4.3: Datasets Discrepancy

4.1.4 Datasets Description

In conclusion, the complete dataset used contains 466 items each represented by 459 features (= 113 exercises * 4 features per exercise + 7 summary features).

In order to evaluate the performance of the various features, four different analyses have been made with sub-sets gradually richer in features:

1. *Attempted* - the dataset is a binary matrix that only considers whether an exercise has been tried by the student at least once or not, regardless of the outcome
2. *Grades* - only the grades obtained by the student in each exercise are taken into account
3. *Grades + Attempts* - the number of times a student has attempted an exercise is also taken into account.
4. *All Features* - is the complete dataset described above

Each of these datasets has been used for the training of the model described in the following paragraph 4.2. The number of features present in the different datasets is reported in the table 4.4.

Dataset	**Features**
Attempted	114
Grades	115
Grades + Attempts	229
All Features	459

Table 4.4: Number of features in each dataset

Finally, all datasets were divided into training and test sets using the *train_test_split* function of *sklearn* using the same *random_state* so that the division is identical for each one. The table 4.5 shows the size of the training and test sets and information on the distribution of labels, i.e. what percentage of the students actually passed the exam.

Dataset	**Samples**	**Passed**	**Passed (%)**
Total	466	357	77%
Training Set	372	290	78%
Test Set	94	67	71%

Table 4.5: Training and Test sets labels distribution for Exercise Quality Analysis

4.2 Model Description

Throughout the project it was decided to use only one classification algorithm in order to compare the performance of the various feature sets.

The choice fell on *Gradient Boosting* and below there are two of its definitions:

> "Gradient boosting is a machine learning technique for regression and classification problems, which produces a prediction model in the form of an ensemble of weak prediction models, typically decision trees. It builds the model in a stage-wise fashion like other boosting methods do, and it generalizes them by allowing optimization of an arbitrary differentiable loss function." (Wikipedia) [9]

> "Gradient Boosted Decision Trees (GBDT) is an accurate and effective off-the-shelf procedure that can be used for both regression and classification problems in a variety of areas" (Scikit-learn) [10].

Gradient boosting [1] is based on three elements:

- a *loss function* that needs to be optimized
- a weak *learner* that makes predictions
- an *additive model* that adds weak learners so as to minimize loss function

The *loss function* must be differentiable and may depend on the type of problem you want to solve. For example, you can use the logarithmic loss function for a classification problem and the squared error for regression.

The *weak learners* used in gradient boosting are the decision trees. They are built in a greedy way by choosing the best split using a purity score (e.g. Gini index) or minimizing the loss function. Usually thresholds are imposed on weak learners, for instance the maximum number of layers, splits or leaf nodes, to make sure that the learners remain weak and easily buildable.

Finally, the *additive model* adds the calculated trees to the model one at a time in order to minimize the loss function with each addition. The training stops when a predefined number of trees have been added or when there are no further improvements in the loss function.

4.2.1 Gradient Boosting Classifier from Scikit-Learn

For this project it was decided to use the *GradientBoostingClassifier* from *scikit-learn*, particularly because of its excellent performance in accuracy and training speed and its overfitting robustness.

Scikit-learn is an open source machine learning library that supports supervised and unsupervised learning. It also provides various tools for model fitting, data preprocessing, model selection and evaluation, and many other utilities [18].

GradientBoostingClassifier is a classification model already developed in the *skelearn.ensemble* library. It has several parameters in order to be able to customize the structure and operation of the model as much as possible, for example:

- *loss* - the loss function that must be optimised.
- *learning_rate* - shrinks the contribution of each tree by its value. There is a trade-off between it and *n_estimators*.
- *n_estimators* - the number of weak estimators generated.
- *max_depth* - maximal depth of each weak estimator.
- *max_leaf_nodes* - grow trees of *max_leaf_nodes* using a best-first approach.

Finally, another reason why this model was chosen is because of the ability to extract the importance of features in model construction using the *feature_importances_* method and thus get a relative idea of which features are more important. The importance of a feature is computed as the normalized total reduction of the criterion brought by that feature.

4.3 Analysis of Results

In this section we will analyze the results obtained for each of the four datasets. To evaluate the outcome of our analysis we decided to use the *classification_report* function in *sklearn.metrics*. It shows, in addition to the accuracy measured on the test set, also the values of Precision, Recall and F1-Score, for each class. Moreover, the confusion matrix of each model has also been reported in order to display how it tends to classify the different entries.

Finally, we have reported, for all datasets, a table showing the 10 most important features. In order to obtain these values we trained each model 1000 times; for each iteration we extracted the vector containing the relevance of each feature thanks to the *feature_importances_* method and added the values obtained for each feature. In this way we can get an idea of which features have a greater weight in model construction.

4.3.1 Attempted

In this first study we were interested in finding out whether a good classifier could be obtained simply by knowing whether or not a student had attempted an exercise.

The model built on the *Attempted* dataset has obtained, as shown in the table 4.6, an accuracy of 0.73. Moreover both the precision and, especially, the recall values, are very low for the *Not Passed* class. In fact, as we can see also from the confusion matrix in figure 4.1 there is a high number of false positives, that is, students who did not pass the exam assigned to the *Passed* class.

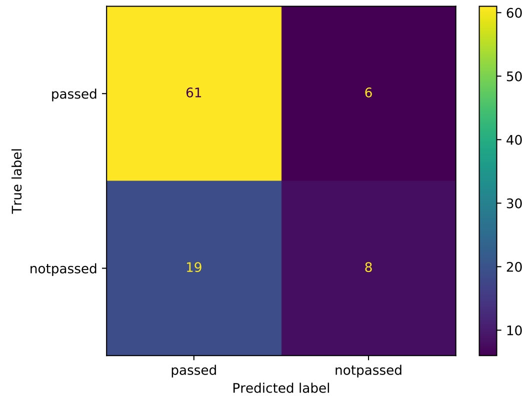


Figure 4.1: *Attempted* Confusion Matrix

	Precision	**Recall**	**F1-Score**	**Support**
Passed	0.76	0.91	0.83	67
Not Passed	0.57	0.30	0.39	27
Accuracy			0.73	94

Table 4.6: *Attempted* classification report

Among the most important features (table 4.7) we can see that *tot_attempted*, i.e. the summary feature that indicates how many exercises have been attempted by a single student, is the one that had by far the most weight. This was predictable as it is the only non-binary feature and therefore contains more information. Furthermore, there may be a positive correlation between the number of attempted exercises and the probability that a student will pass the exam at the end of the semester.

Attempted	
Feature	**Score**
tot_attempted	255.62
StudentInit_attempted	68.16
MergeList_attempted	60.34
Bath_attempted	26.01
DebtReminder_attempted	25.82
TextToDic_attempted	25.60
SimpleMath_attempted	22.89
REAL05_attempted	20.79
Prime_attempted	20.21
LinkedListEndRemove_attempted	19.58
SUM	545.00

Table 4.7: *Attempted* Features Importance sum after 1000 iterations

4.3.2 Grades

In this second analysis we used the students' grades for each exercise instead. Surely they will provide more information than just counting the attempted exercises.

The model built on the *Grades* dataset has obtained, as shown in the table 4.8, an accuracy of 0.74, just a bit better than the previous one. As in the first analysis, both the precision and the recall values, are very low for the *Not Passed* class. In fact, the confusion matrix in figure 4.2 there is still a high number of false positives.

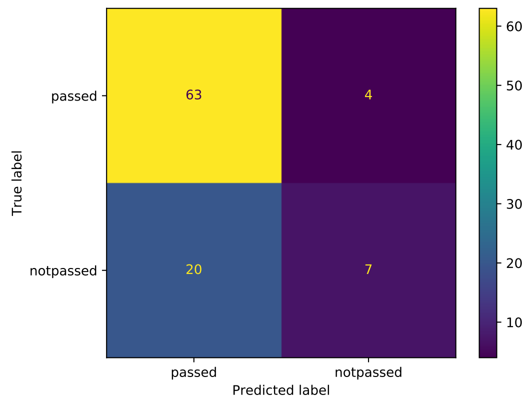


Figure 4.2: *Grades* Confusion Matrix

	Precision	Recall	F1-Score	Support
Passed	0.76	0.94	0.84	67
Not Passed	0.64	0.26	0.37	27
Accuracy			0.74	94

Table 4.8: *Grades* classification report

As we can see in the table 4.9 again, the summary features *avg_best_grade_total* and *avg_best_grade_tried* are the ones that received the most attention. In fact, as in the previous case, it can be deduced that the students with the highest grade average have the best chance of passing the exam. Finally, as we can see from the *SUM* field, the first 10 features continue to represent over 50% of the total importance.

Grades	
Feature	**Score**
avg_best_grade_total	148.22
avg_best_grade_tried	71.73
Session6_QCM_best_grade	49.61
REAL05_best_grade	41.24
StudentConstructor_best_grade	39.38
DebtReminder_best_grade	36.93
Session1_QCM_best_grade	35.49
StudentInit_best_grade	33.39
SimpleMax_best_grade	25.40
Count_best_grade	25.15
SUM	506.57

Table 4.9: *Grades* Features Importance sum after 1000 iterations

4.3.3 Grades + Attempts

In the third study we added features on the number of attempts made for each exercise, assuming that a student who makes fewer attempts is better at solving an exercise.

The model built on the *Grades + Attempts* dataset has obtained another slight increase in accuracy, reaching 0.76 (see table 4.10), but as in the previous cases we still have a high number of false positive entries as we can see also from the

confusion matrix in figure 4.3.

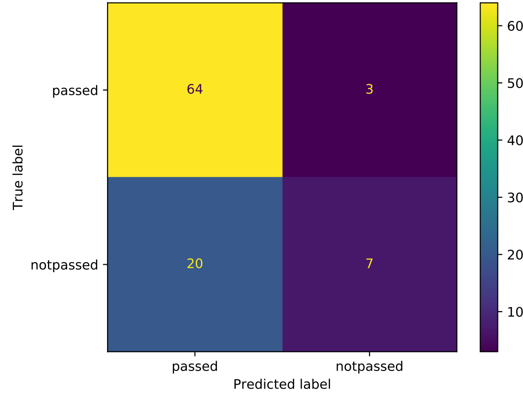


Figure 4.3: *Grades + Attempts* Confusion Matrix

	Precision	**Recall**	**F1-Score**	**Support**
Passed	0.76	0.96	0.85	67
Not Passed	0.70	0.26	0.38	27
Accuracy			0.76	94

Table 4.10: *Grades + Attempts* classification report

In this case we can see that *avg_best_grade_total* is still a feature with some importance, but not the same as the summary features in the previous examples. In general, however, features that indicate the number of attempts on specific exercises seem to be more important than features that indicate the grade obtained.

Grades + Attempts	
Feature	**Score**
Session4_QCM_attempts	64.17
AmazonConstructor_attempts	54.00
avg_best_grade_total	50.17
DiffCount_attempts	38.94
Coordinates_attempts	38.63
GCD_attempts	29.60
Session6_QCM_best_grade	27.85
SimpleMax_best_grade	27.71
MergeList_attempts	26.88
StudentInit_attempts	26.77
SUM	384.76

Table 4.11: *Grades + Attempts* Features Importance sum after 1000 iterations

4.3.4 All Features

Finally, the model built on the *All Features* dataset is the one that has obtained, as shown in the table 4.12, the best accuracy score of 0.78. As all the previous models, both precision and recall scores, are quite low for the *Not Passed* class. In fact, as we can see also from the confusion matrix in figure 4.4 there is still a consistent number of false positives.

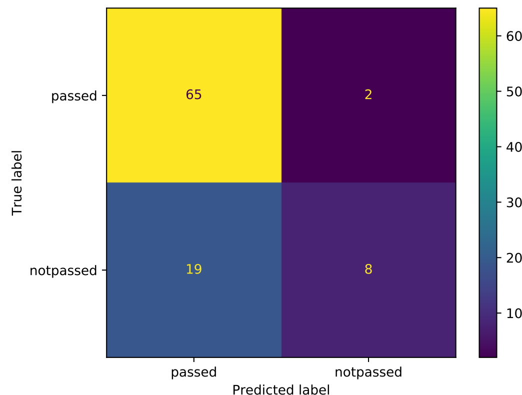


Figure 4.4: *All Features* Confusion Matrix

	Precision	**Recall**	**F1-Score**	**Support**
Passed	0.77	0.97	0.86	67
Not Passed	0.80	0.30	0.43	27
Accuracy			0.78	94

Table 4.12: *All Features* classification report

In this case the only summary feature that is in the top 10 is *avg_days_since_beginning* and in general the time passed since the beginning of the semester seems to be a type of feature that provides enough information about the final result of the exam. We can also notice a big decrease in the sum of the importance values of the first 10 features, but we must also take into account that the number of features is much higher than the other three datasets.

All Features	
Feature	**Score**
AmazonConstructor_attempts	56,50582
FirstSum_time_to_best_grade	41,84725
avg_days_since_beginning	40,74566
Session1_QCM_days_since_beginning	38,1318
Prime_days_since_beginning	32,16751
Coordinates_attempts	26,93347
MergeList_days_since_beginning	23,26265
Prime_time_to_best_grade	19,94707
Session4_QCM_attempts	19,62833
StudentInit_days_since_beginning	18,29672
SUM	317,46628

Table 4.13: *All Features* Features Importance sum after 1000 iterations

4.4 Final Considerations

In general, all four models performed similarly, although as the number of features increased, the results were progressively better. Unfortunately, the classifiers seem to be quite inclined to include students in the class *Passed*. This may be due to the fact that over 75% of the students considered have passed the exam and therefore the dataset is unbalanced. Also keep in mind that the sample size is quite limited,

only 466 students, which even leads, in the case of *All Features*, to have almost as many features as entries.

In conclusion, it would be interesting to repeat this analysis on a larger sample and maybe selecting sub-sets of exercises considered more important by the teachers. This analysis was not deepened because it was only meant to be taken as a starting point for the real core topic of the project that will be described in the next chapters.

Chapter 5

Preparation for Source Code Analysis

In this chapter we will describe all the steps that were necessary to generate the data for each of the source code analysis described in the next chapter.

It will be divided into the following sections:

- *Exercise Selection* - in which we will briefly describe the process of selecting exercises for source code analysis.
- *External Resources* - where will be described the two tools used for this study: *astminer* and *code2vec*, with particular attention to the latter which is a cornerstone of all this research.
- *Data Description* - where we'll talk about the extraction of the data from the database, the preprocessing steps that were necessary to make the entire pipeline work and the choices on how to split the datasets.
- *Code2Vec Execution* - in which we will describe the way we run *code2vec*.
- *Data Postprocessing* - where we talk about the final steps to obtain the files needed for our studies.

5.1 Exercise Selection

For this analysis it was decided not to use all the 113 exercises proposed in the Syllabus described in chapter 3.1, as many of them were not considered interesting or suitable for this research.

First of all, all multiple-choice exercises have been excluded, as they do not involve the writing of source code by the student. In addition, several exercises, especially those concerning the first topics, which are very simple and require only

a few lines of code to be written, have been discarded too since they are not presented as a method. Indeed, as we will see in the chapter 5.2.2, *code2vec* works on individual methods, so exercises with few lines of code and without a real context have been discarded.

Finally, for the same reason mentioned above, exercises that are too long and complex have also been excluded, for example, those in which you have to declare various classes and methods, which would then be unpacked into several entries for *code2vec* and lose their overall meaning.

At the end of this selection process there were only 40 exercises left that were considered suitable for this task. Probably with a more detailed analysis a larger number could have been selected, but in order to avoid case-by-case evaluations it was decided to exclude the exercises for which a stricter control would have been necessary.

5.2 External Resources

5.2.1 Astminer

Astminer [12] is a library used for the mining of path-based representations of code and more. It is supported by the "Machine Learning Methods for Software Engineering" group at JetBrains Research [4].

In our project, it has been used, together with other preprocessing steps that will be described in the chapter 5.3.2, to convert the source code extracted from the database into a format compliant with the one required by *code2vec*. It was also used to save the ASTs in *DOT* format so that the structure of the trees generated by the code snippets can be displayed.

We used the CLI version of the tool, integrating it into a script that performs the various stages of preprocessing. Below is reported the pseudo command to run the tool.

```
java -jar cli.jar code2vec --lang py,java,c,cpp
    --project path/to/project --output path/to/results
```

As we see, it receives four parameters:

- *code2vec* - specifies that we are interested in the output format compatible with the model *code2vec*.
- *–lang* - must be followed by one of the four supported programming languages (Python, Java, C and C++) to indicate the type of files that will be passed as input.
- *–project* - is followed by the path to the input folder, all files in the specified folder and its subfolders will be processed.

- *–output* - is followed by the path where we want the files generated by *astminer* to be saved.

We show an example of the use of *astminer* passing as input a project containing only one *findMax* method. We report the code below.

```
def findMax(vet):
    maximum = vet[0]

    for num in vet:
        if num > maximum:
            maximum = num

    return maximum
```

As we said *astminer* can also generate the Abstract Syntax Tree of a method in *DOT* format. Below we show two AST for the *findMax* method: the raw one generated by *astminer* (Figure 5.1) and the a more interpretable one made by us with a specific conversion script (Figure 5.2)

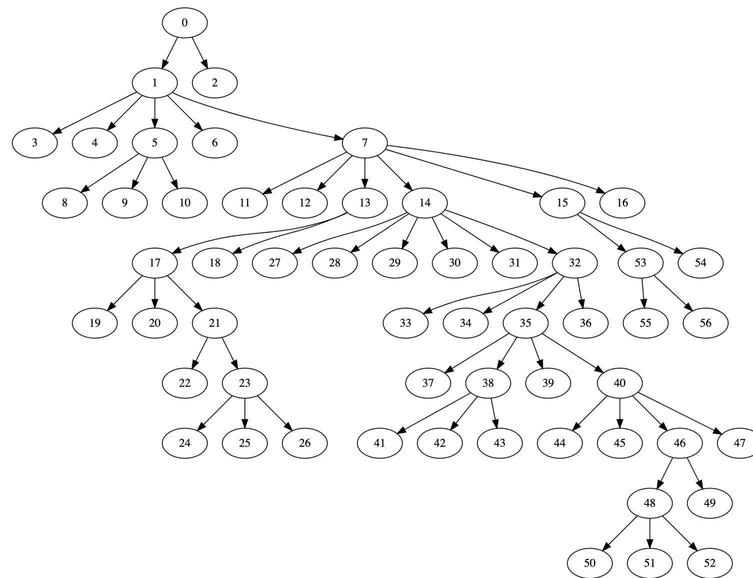


Figure 5.1: *findMax* AST generated by astminer

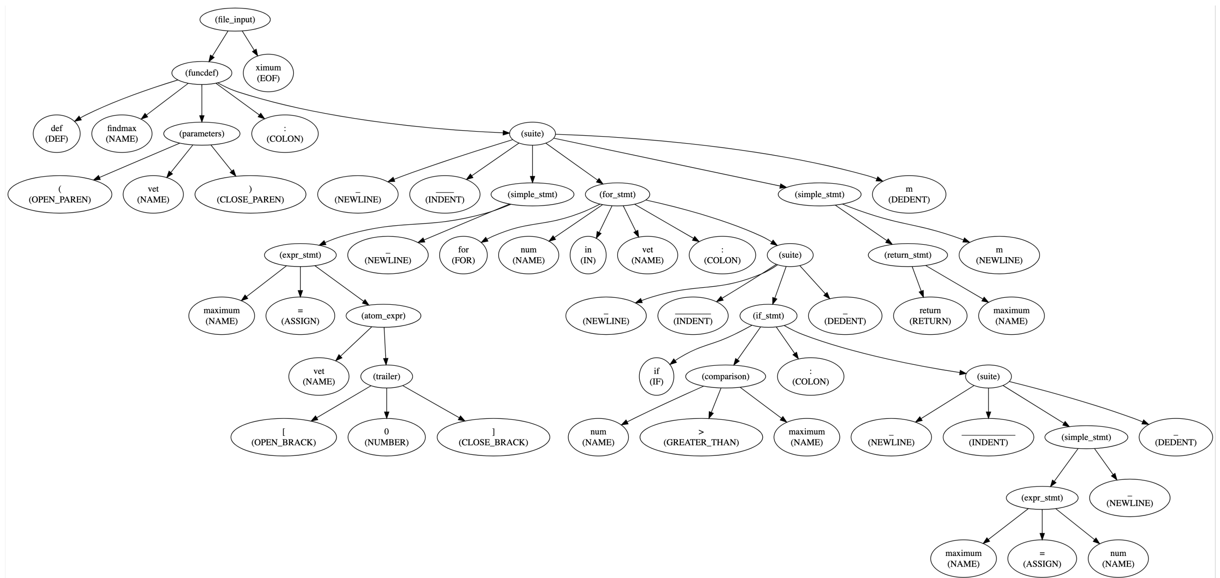


Figure 5.2: *findMax* AST with labels

Astminer processes all files within the project passed as input and returns four *csv* files as output:

- *tokens.csv* - indexes the tokens, i.e. all the leaf nodes of the AST (See table 5.1).

id	token
1	(
2	vet
3	)
4	[
5	0
6	]
7	maximum
8	=
9	_
10	num

Table 5.1: Sample of entries in tokens.csv

The tokens can be considered as labels for the leaf nodes of the AST. They can be symbols (such as parentheses or mathematical signs), but also the names assigned to variables (e.g. *vet* or *maximum*) and the values assigned to them (e.g. *0*).

- *node_types.csv.* - indexes all the different node types present in the AST extracted from the input files (See table 5.2).

id	node_type
1	OPEN_PAREN UP
2	parameters UP
3	parameters DOWN
4	typedargslist|tfpdef|NAME DOWN
5	CLOSE_PAREN DOWN
6	typedargslist|tfpdef|NAME UP
7	OPEN_BRACK UP
8	trailer UP
9	trailer DOWN

Table 5.2: Sample of entries in node_types.csv

The *node_types* are instead, as the name says, the type of node in the tree. *Astminer* tends to generate very long nodes separated by the "|" symbol, so in graphical representations we decided to take only the last part of a node (example from "typedargslist|tfpdef|NAME UP" we take only "NAME UP"). In addition each node ends with "UP" or "DOWN" which indicates whether we are moving up or down within the AST.

- *paths.csv.* - indexes all the paths extracted from the ASTs. A path is a sequence of nodes connected to each other (See table 5.3).

id	path	path (corresponding node_types)
1	1 2 3 4	(OPEN_PAREN)^(parameters)^(parameters)_(NAME)_
2	1 2 3 5	(OPEN_PAREN)^(parameters)^(parameters)_(CLOSE_PAREN)_
3	6 2 3 5	(NAME)^(parameters)^(parameters)_(CLOSE_PAREN)_
4	7 8 9 10	(OPEN_BRACK)^(trailer)^(trailer)_(NUMBER)_
5	7 8 9 11	(OPEN_BRACK)^(trailer)^(trailer)_(CLOSE_BRACK)_

Table 5.3: Sample of entries in paths.csv + paths as string

A path is the sequence of nodes connecting two leaf nodes. *Astminer* to save memory stores the nodes with their indexes, but in the table we also showed a more interpretable version where we reported the *node_types* and replaced "UP" and "DOWN" with the symbols "^" and "_" respectively.

- *path_contexts.csv.* - contains as many lines as the methods extracted from the input files. Each line contains the method name and then a sequence of

path_contexts extracted from that method (See table 5.4)

Method Name	**Path Contexts**
find|Max	1,1,2 1,2,3 2,3,3 4,4,5 4,5,6 ...

Table 5.4: Content of path_contexts.csv

Each *path_contexts* consists of three elements: the starting token within the AST, the ending token and the path that connects them. They are represented as a triplet in this order: *start_token, path, end_token.* For example the *path_contexts* 4,4,5 has the following components:

- *start_token* : [
- *path* : (OPEN_BRACK)^(trailer)^(trailer)_(NUMBER)_
- *end_token* : 0

It is precisely the data contained in the file *path_contexts* that *code2vec* takes as input in order to generate a predictive model and extract the vector representations of the code.

5.2.2 Code2Vec

In this section we will try to go into more detail, compared to what was said in chapter 2.2.1, on how *code2vec* was implemented.

Background

To better understand how this algorithm works, we need to define three recurring elements:

- *Abstract Syntax Tree* (AST) - as described several times is a way of representing code as a tree composed of intermediate nodes (representing abstractions such as the definition of a function, constructs *if*, *for* or *while*, but also the declaration and assignment of values to a variable) and leaf nodes that instead associate with abstractions what is actually written in the source code.

- *AST Path* - is a sequence of nodes that join two leaf nodes, which also indicates the direction (up or down) to go from one node to the other.

- *Path Context* - is a triplet $< x_s, p, x_t >$, where p is the *AST Path* while x_s and x_t are the values associated with the start and end nodes respectively.

Model

The principle behind *code2vec* is the idea that a source code snippet can be seen as a bag of *path_contexts*. Each of these *path_contexts* is represented by a vector that must capture its semantic meaning, but also the amount of attention to be given to itself. In fact, there may be *path_contexts* common to all methods (e.g. the definition of the method itself) that therefore do not give relevant information about the code and consequently do not need to receive much attention.

The main problem is to aggregate the *path_contexts* of a code snippet into a single vector. A trivial approach could be to take the most important *path_context* or average all of them, but, as we have already said, a single *path_context* cannot contain all the properties of a method and, at the same time, not all *path_contexts* have the same importance. The goal is therefore to use them all, but the model must also be able to understand how much attention to give to each of them. All this is done by making the dot product of the *path_contexts* vectors with an *attention* vector. All these vectors are learned simultaneously using the classic backpropagation method.

Code2Vec uses the following components:

- *path_vocab* - the matrix containing the embeddings for all paths, each line represents a *AST Path*.

  $$path_vocab \in \mathbb{R}^{|P| \times d}$$

  where P is the set of *AST Paths* and d the is the embedding size. d is determined empirically limited by the training time, the model complexity and the GPU memory, usually it is between 100 and 500.

- *value_vocab* - the matrix representing embeddings for tokens, each line represents a token.

  $$value_vocab \in \mathbb{R}^{|X| \times d}$$

  where X is the set of AST terminal nodes observed during training.

- W - the matrix of weights for the fully connected layer

  $$W \in \mathbb{R}^{d \times 3d}$$

- $\boldsymbol{a}$ - the attention vector

  $$\boldsymbol{a} \in \mathbb{R}^{d}$$

- *tags_vocab* - the matrix containing the embedding for the labels

$$tags_vocab \in \mathbb{R}^{|Y| \times d}$$

where Y is the set of labels found in the training corpus.

The values of all components are randomly initialized and learned during training.

As we said, *code2vec* receives a series of *path_contexts* related to a method, they are triplets $< x_s, p_j, x_t >$ where $\{x_s, x_t\} \in X$ and $p_j \in P$. The three embeddings are concatenated forming a single *context vector*: $\boldsymbol{c_i} \in \mathbb{R}^{3d}$

$$\boldsymbol{c_i} = embedding(< x_s, p_j, x_t >) = [value_vocab_s; path_vocab_j; value_vocab_t] \in \mathbb{R}^{3d}$$

Then, a fully connected layer learns to combine its components. This process is done separately for each context vector and this allows the model to give different attention to different combinations of paths and tokens. In fact the same path can receive different attention values depending on the tokens it connects, receiving more importance in one case and less in another.

The *combined context vector* $\tilde{c_i} \in \mathbb{R}^d$ is obtained through the fully connected layer represented by the W matrix in the following way:

$$\boldsymbol{\tilde{c_i}} = tanh(W \cdot \boldsymbol{c_i})$$

where *tanh* is the hyperbolic tangent function, commonly used as a nonlinear monotonic activation function that returns values between $(-1, 1)$, which improves the expressiveness of the model. The fully connected layer allow to compress a vector of size $3d$ into one of size d.

The attention mechanism is used to calculate the weighted average of the various *combined context vectors* by calculating the scalar weight of each of them.

The attention weight α_i of each $\boldsymbol{\tilde{c_i}}$ is computed as the normalized inner product between the *combined context vector* $\tilde{c_i}$ and the *global attention vector* $\boldsymbol{a}$:

$$attention\ weight\ \alpha_i = \frac{exp(\boldsymbol{\tilde{c_i}}^T \cdot \boldsymbol{a})}{\sum_{j=1}^{n} exp(\boldsymbol{\tilde{c_i}}^T \cdot \boldsymbol{a})}$$

The *aggregated code vector* $\boldsymbol{v} \in \mathbb{R}^d$, which represents the whole code snippet, is a linear combination of the combined context vectors $\{\boldsymbol{\tilde{c_1}}, ..., \boldsymbol{\tilde{c_n}}\}$ factored by the attention weights:

$$code\ vector\ \boldsymbol{v} = \sum_{i=1}^{n} \alpha_i \cdot \boldsymbol{\tilde{c_i}}$$

the attention weights are non-negative and their sum is 1, so attention can be seen as a weighted average, where weights are learned and calculated compared to all *path_contexts*.

Finally, the prediction of the tag is performed using the *code vector*. *Tags_vocab* contains all the labels seen during the training phase. The predicted distribution of the model $q(y)$ is computed as the softmax-normalized dot product between the *code vector* $\boldsymbol{v}$ and each of the tag embeddings:

$$for\ y_i\ in\ Y : q(y_i) = \frac{exp(\boldsymbol{v}^T \cdot tags_vocab_j)}{\sum_{y_j \in Y} exp(\boldsymbol{v}^T \cdot tags_vocab_j)}$$

The probability that a specific tag y_i should be assigned to the the given code snippet C is the normalized dot product between the vector of y_i and the *code vector* $\boldsymbol{v}$.

For training the network they use the cross-entropy loss [17] between the predicted distribution q and the "true" distribution p, where p is a distribution that assign the value 1 to the actual tag in the training sample and 0 otherwise.

Once the model is trained it can be used in two ways:

- use the *code vector* $\boldsymbol{v}$ for another task down the line.
- use the learned template to predict labels for unseen code.

As we will see in chapter 6, for our datasets we have used the *code vectors* to train a *gradient boosting* classifier in order to execute our analysis.

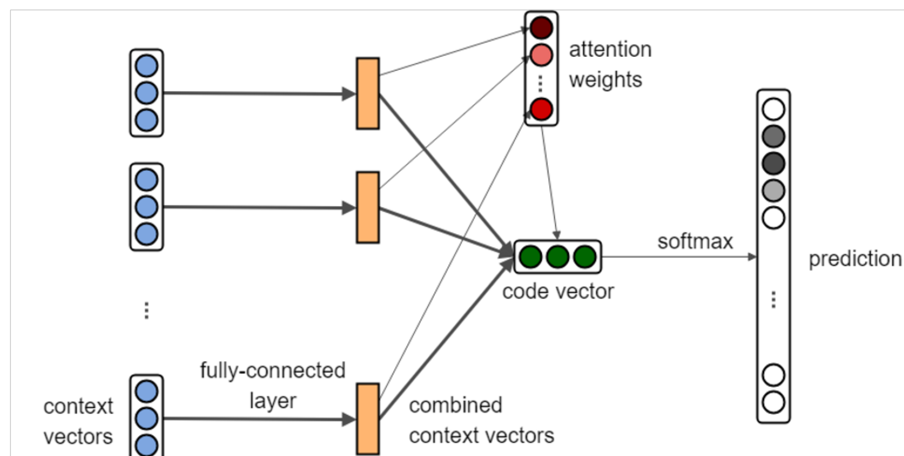


Figure 5.3: *Code2vec* model architecture

Changes for our purpose

In their implementation of the code [8] Alon et al. provide all the tools to preprocess a java project dataset and run the model training, then it is possible to evaluate its performance on a test set or run the model interactively on a single java file to examine its prediction and attention scores.

In addition, it is possible to extract vectors representing tokens and target labels and export the code vectors.

For our purposes we were interested in extracting both vector representations and attention scores from an entire set of methods passed to the model, so that we could use those data in our *Gradient Boostring* model.

To do this, we implemented a custom command called *my_predict*. It is able to generate, given a set of methods, a *JSON* file that contains all the data we are interested in for each of the methods passed as input.

5.3 Dataset Description

In this project we used five datasets: one containing all the exercises done during the year by all the students, while the other four contain each the code written by the students for questions 1, 2, 3 and 6 of the January 2019 exam. Questions 4 and 5 were discarded because:

- question 4 required the development of two methods and, as we have seen, *code2vec* separates different methods into two separate entries, so it would have been complex to re-associate both methods to the same exercise.
- question 5 instead involved the writing of several classes and *code2vec* is not yet able to handle this kind of structure.

5.3.1 Data Extraction

For the selection of the files we followed an approach similar to the one discussed in chapter 4.1.1. In this case, we took for each student the source code of each exercise he did by selecting the most recent and highest grade submission. In fact, a student may upload a solution that gets a 100% grade several times, because he decides to make some minor tweaks to optimise or clean up the code.

Each file is saved in the database in the bucket *fs* within a BSON type file [6] (i.e. a binary JSON). The BSON file is converted into a dictionary within which there are some parameters (such as the student name) and a string containing the source code.

It is then generated for each exercise and each student a file *.py* containing the source code whose name has the following format: <*task_name*>___<*student_name*>.py.

So that you can later keep track of the association between the source code and the student who wrote it.

In the table 5.5 we show the size of each dataset.

Dataset Name	**Dimension**
exercises	16447
0119_q1	606
0119_q2	568
0119_q3	562
0119_q6	470

Table 5.5: Dimensions of datasets for source code analysis

During the extraction of student files, other files are also generated:

- the final result of the exam (*Passed* or *NotPassed*), where the exam is considered passed if the student has obtained a score greater than or equal to 10 out of 20.
- the outcome of individual examination questions (*Passed* or *NotPassed*), where the question is considered sufficient if it obtains a grade greater than 50%.

These will be used as the labels to be predicted in the analysis that we will describe in the next chapter.

5.3.2 Data Preprocessing

To go from extracting files from the database to running *astminer* and finally using *code2vec*, a few data preprocessing steps are required.

These steps are the same for each dataset used and each analysis performed.

As we have seen in the chapter 5.2.1, *astminer* generates, in addition to the files containing: tokens, node_types and paths, the files *path_contexts* in which is reported, in each line, the name of every method contained in the input folder and a series of *path_contexts* extracted from the method itself.

In this way, however, we lose any reference to the student and the exercise to which the source code corresponds. This reference can be very useful later to interpret the results obtained.

Moreover, unlike what is described in chapter 2.2.1, we are not interested in training the model to predict the name of the method as done by Alon et al. [2], but rather to see if, given the source code of an exercise, we are able to understand whether or not the student will pass the exam. In fact we use *code2vec* as a binary classifier.

For this reason, our first preprocessing step is to replace the method name within each file with a unique string (*key_string*) and keep track of the information about it in another file, such as the student name, the task name and the *Passed* or *NotPassed* label that will be assigned to it later.

Once this is done we can run *astminer*. In order to parallelize the process, it will generate several *path_contexts* files which will contain one line for each recognized method. It may happen that a student defines several methods inside each other, in these cases they will be considered as separate methods, but with the same *key_string*. This may generate duplicates, as the "father" method will also contain the *path_contexts* of the inner methods. To avoid this, only the first instance (that of the "father" method) is taken for each *key_string*, while the others are discarded. This selection process is done when we merge the various *path_contexts* files into a single one.

5.3.3 Data Split

Finally, two sets of files are generated to create the two models (*code2vec* and *GradientBoosting*) used in this analysis:

- For *code2vec* three sets are created: *training*, *validation* and *test*; containing 70%, 10% and 20% of the samples respectively.

- For *GradientBoosting* the same *test* set of *code2vec* is used, while for training the union of *training* and *validation* sets described before.

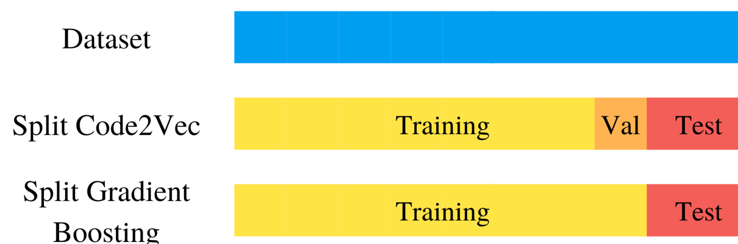


Figure 5.4: Dataset split

The first files are used to generate the model *code2vec* using *Passed* and *NotPassed* as the label. Once this model is built, the *my_predict* function described in chapter 5.2.2 is used to get the code vectors and attention values of the second set of files, where the labels are the *key_strings*. In this way we are able to associate the code vector and the source file from which it was generated.

5.4 Code2Vec Execution

Once all the steps described so far have been done, we can use the scripts provided in the *code2vec* project. In fact before executing the training of the model it is necessary to execute a last step of preprocessing that, starting from the files that we have generated as described in the previous section, creates the files that will actually be taken as input by the model.

For our research we had to slightly modify this preprocessing script to adapt it to our purposes. Both sets described in chapter 5.3.3 are converted.

Afterwards, the model training is done. We decided to leave all the default parameters apart from the number of training periods (*NUM_TRAIN_EPOCHS*). Since our datasets are much narrower than the one used by Alon et al. [2] we can afford and do the training for a larger number of epochs, although the algorithm is set to stop earlier if the results do not improve.

The model at each epoch uses: the *training* set to learn all the embeddings and the weight matrix described in chapter 5.2.2 and then the *validation* set to evaluate the improvements by calculating the *F1-Score*.

Once the training is finished the model is tested on the *test* set to get an additional performance indicator.

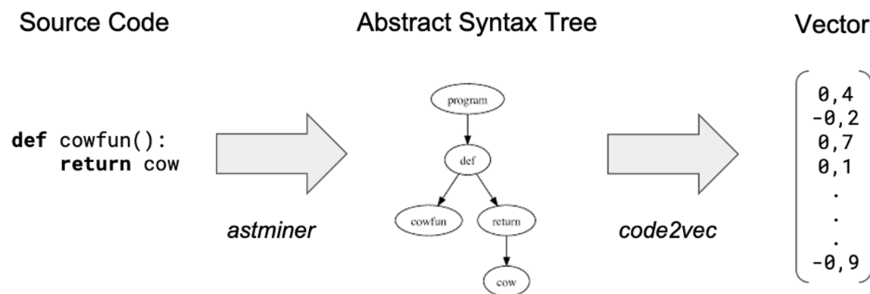


Figure 5.5: Steps of source code conversion

5.5 Data Postprocessing

Finally, it's called the *my_predict* method. It receives as input the *training* and *test* files generated for the *GradientBoosting* model and for each of them it generates a *JSON* file containing a dictionary with *key_string* as its primary key and the following fields:

- the *prediction* of the model, i.e. what percentage of the method belongs to the *passed* and *notpassed* classes.

- the *code_vector*, a vector of 384 values representing the method itself.
- the dictionary of *attentions* that lists the 10 *path_contexts* that received the most attention for the prediction.

This file is later integrated in the post processing phase by adding also:

- the name of the student
- the name of the exercise you have carried out
- the original name of the method
- the actual label assigned to it

Below is the structure of one of these final *JSON* files. They will be used for all the analyses described in the next chapters.


```
{
    "bcaaa": {
        "predictions": [
            [
                0.6132161617279053,
                "passed"
            ],
            [
                0.38403835892677307,
                "notpassed"
            ]
        ],
        "attentions": [
            {
                "score": 0.047637175768613815,
                "path":
                    "(NEWLINE)^(simple_stmt)^(suite)^(suite)_(DEDENT)_",
                "token1": "_",
                "token2": "_"
            },
            ...
        ],
        "code_vector": [
            -0.11657661944627762,
            -0.12590880692005157,
            -0.17035745084285736,
            0.19726648926734924,
            ...
        ],
```

```
        "student": "gallezt",
        "task_name": "0119_q1",
        "method_name": "faulhaber_max",
        "exam_outcome": "passed"
    },
    ...
```

In the figure 5.6 we can see a scheme that summarize all the steps that we carry out for our analysis.

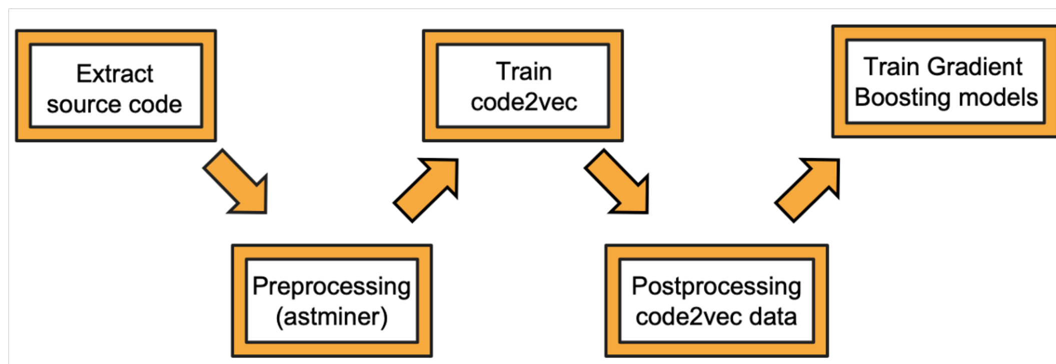


Figure 5.6: Pipeline steps

Chapter 6

Analysis on Exercise Source Code

In the course of our research we have carried out various types of experiments to determine whether, through the analysis of the source code, interesting and useful results can be obtained for the evaluation of the exercises done by the students.

In this chapter we will describe the results obtained from the various analyses executed on the source code. It will be divided into the following sections:

- *Type of Models* - in which we will briefly describe the two models we trained for each dataset.
- *Exercises on Exams Outcomes* - where we will try to predict the result of a student's examination by locking at the source code of the exercises uploaded during the course.
- *Exercises on Questions Outcomes* - where we will report the results of trying the predict the outcome of a specific question of the exam starting from the exercises developed throughout the semester.
- *Exam Questions Outcome* - in which we will report whether or not the evaluation of the raw source code is enough to estimate the quality of an exam question.
- *Final Considerations* - where we will comment the results obtained and discuss the limits and possible improvements.

6.1 Type of Models

For each analysis we have built two different models from the data extracted at the end of the whole pipeline described in chapter 5.

Both models are based on the *GradientBoostingClassifier* described in the section 4.2, the difference is in the way we decided to represent each method that is given as input to the model.

6.1.1 Model on Code Vector

The first model is trained by inputting the vector representations generated by *code2vec*, where each method is represented by an array of 384 values.

This number is the d value described in chapter 5.2.2, which is the size of all the embeddings generated by *code2vec*. The value can be chosen arbitrarily, although it has been decided to keep the default one used by the code authors.

As in all *embedding* methods, the larger the size of the object's representative vector, the more information it contains, and the goal is to find the right compromise between these two factors, since a vector with many features takes longer to be evaluated.

These vectors, although they often manage to capture even very subtle differences and similarities among the methods, are often difficult to interpret.

In the case of *code2vec* all the elements used by it (*paths*, *tokens*, *path_contexts*, *labels*) are represented in vector form starting from randomly initialized values and gradually learned through error back propagation techniques. In addition, a fully connected layer is used to aggregate all the *path_contexts* of a method. All this makes it impossible to interpret these vectors and to extract information from them afterwards. For this reason we have chosen to try a second representation.

6.1.2 Model on Paths Vector

In the second model we decided to represent each method as a binary vector of *paths*. Each value of the vector represents a *path* and is assigned the value 1 if that *path* is present among those contained in the *path_contexts* extracted by *astminer* and 0 otherwise.

To determine which *path* to take into account we decided to use the output obtained from our pipeline as a reference. In fact, as described in the last chapter *code2vec* is able to extract for each method the ten *path_context* considered most relevant for the classification.

On this basis we extracted, for each of the top 10 *path_context* of each method, the *paths* and counted the recurrences of each of them. Finally we sorted them in order of recurrence. In this way we use *code2vec* as a kind of filter that extracts the most important paths among all the methods.

From this ordered list we extract the identifiers of the first 100 *paths* and build for each method a binary vector as described above.

We tested different lengths for this vector, between 50 and 500, without any particular difference in performance and decided to use the smallest value that

gave the best accuracy.

We also tested a non-binary vector that would also track the number of times a given path appeared in the code, but even this did not lead to substantial improvements in the results.

This model, unlike the previous one, is more easily interpreted retrospectively and can provide interesting information on which *path* have the most influence on the result.

6.2 Exercises on Exams Outcomes

In our first analysis we wanted to get familiar with *code2vec* and test how it works.

The idea is to assess whether, given an exercise carried out by a student, it is possible to understand whether or not the student will pass the final exam.

6.2.1 Dataset

The dataset used is *exercises* containing all the best submissions of each student for each of the 40 exercises selected according to the criteria described in the section 5.1.

Each exercise has been assigned a label depending on whether the student passed (*Passed*) or failed (*NotPassed*) the exam in one of three sessions in 2019. Below there is a table (6.1) with general information about the dataset.

Dataset	**Samples**	**Passed**	**Passed (%)**	**Tokens**	**Paths**
Exercises	16447	9810	59,6	2856	22738

Table 6.1: Exercises on exam outcome dataset information

It was then divided, as described in the section 5.3.3, into three sets (table 6.2):

- *training* and *validation* - used to train the model *code2vec* and then combined to train the two *gradient boosting* models described in the section 6.1.
- *test* - used to evaluate the performance of each of both models.

Dataset	Samples	Passed	Passed (%)
Training	11512	6845	59,5
Validation	1645	983	59,8
Test	3290	1982	60,2
Total	16447	9810	59,6

Table 6.2: Exercises on exam outcome data split

6.2.2 Analysis on Code Vector

As we can see from the confusion matrix in the figure 6.1 and the results report in the table 6.3. This model fails to classify with great precision if from the code of a single exercise it is possible to predict the result of the examination.

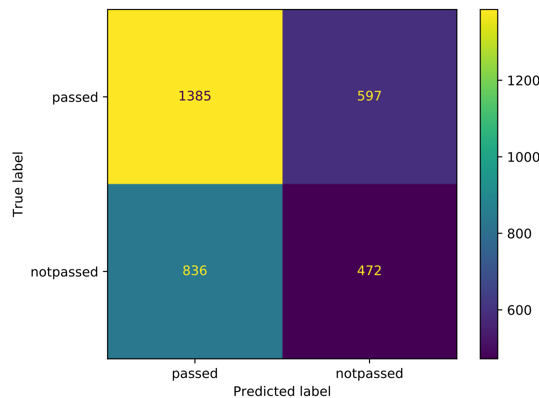


Figure 6.1: Confusion Matrix for *Exercises on exam outcome* using model on *code vectors*

We can notice that all four classes of the confusion matrix have a relatively homogeneous distribution, but in particular we notice the presence of many *False Positive* elements.

This may be due to the fact that a student could have done some exercises correctly, maybe the easier ones, over the course of the year, but may have failed the exam in the end.

	Precision	Recall	F1-Score	Support
Passed	0,63	0,68	0,65	1982
Not Passed	0,45	0,4	0,42	1308
Accuracy			0,57	3290

Table 6.3: Classification Report for *Exercises on Exam outcome* using model on *code vectors*

The model has an accuracy of only 0.57, slightly better than random guessing, but still not enough to say it is effective.

Actually the task required to the model is very complex and generic as it should predict the final result of a student's exam simply starting from a single exercise regardless of the difficulty of it and how many of them the students have done altogether.

6.2.3 Analysis on Path Vector

Although based on accuracy alone, it may seem that the *path vectors* model performs better than the previous one, by looking at the confusing matrix in the figure 6.2 it is clear that the model is unable to classify the source code and consequently it labels everything with the most common label (*Passed*).

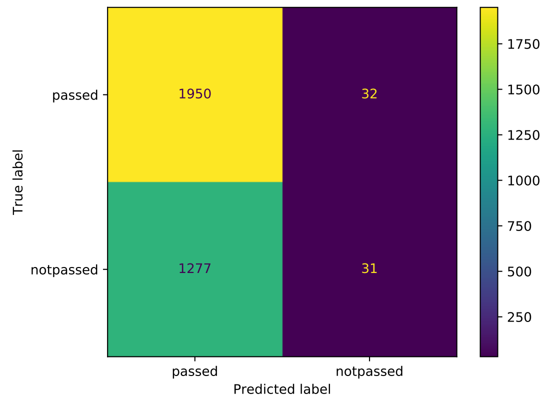


Figure 6.2: Confusion Matrix for *Exercises on exam outcome* using model on *path vectors*

What we said above can be easily seen even when looking at the classification report in the table 6.4, where although the *precision* values may seem good, the *recall* makes us notice the same problem.

	Precision	**Recall**	**F1-Score**	**Support**
Passed	0,6	0,98	0,75	1982
Not Passed	0,48	0,02	0,05	1308
Accuracy			0,6	3290

Table 6.4: Classification Report for *Exercises on Exam outcome* using model on *path vectors*

Since the *exercises* dataset, as we have shown in the table 6.1, contains many paths, we thought that 100 might not be enough to differentiate the exercises and we ran tests using *path vectors* containing up to 5000 of most frequent paths, but without being able to improve the quality of the results.

Finally, analyzing confidence (i.e. how likely an entry is assigned to each of the two labels) we noticed that the average of this value is 0.59. This indicates that the model is very undecided about which class to assign a code snippet to, and in the rare cases where one of them is assigned to the negative class, it is for confidence values extremely close to 0.5.

6.3 Exercises on Exam Questions Outcomes

Given the results of the first analysis we wanted to test our models on a task that was considered easier.

Instead of finding out whether or not the student who has done an exercise will pass the entire exam, we tried to narrow the field by checking whether or not, given an exercise, it is possible to understand if the student will pass a certain exam question.

6.3.1 Dataset

We did this analysis for each of the four questions selected from the January 2019 exam (*0119_q1*, *0119_q2*, *0119_q3*, *0119_q6*).

For the construction of the models we have still used the dataset *exercises*, but each time we assigned the label corresponding to whether or not each of the four questions was passed by the student who has done the exercise. In the table 6.5 we see the information related to the four datasets generated.

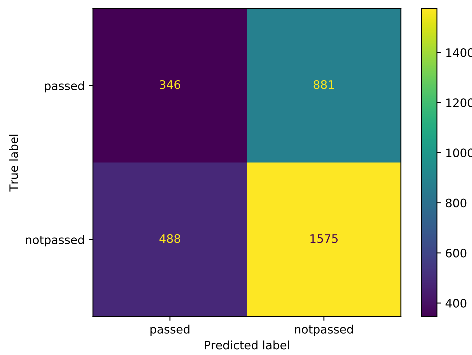
Dataset	Samples	Passed	Passed (%)	Tokens	Paths
Exercises_q1	16447	6081	37,0	2856	22738
Exercises_q2		5706	34,7		
Exercises_q3		2739	16,7		
Exercises_q6		2118	12,9		

Table 6.5: Exercises on exam questions outcome datasets information

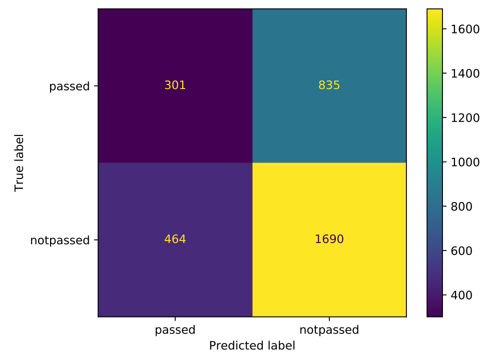
Obviously the only values that change are those related to the number of label *Passed* and we see that this percentage tends to decrease question after question, in fact it is obvious that the questions become increasingly complex and therefore there are fewer students able to answer correctly.

6.3.2 Analysis on Code Vector

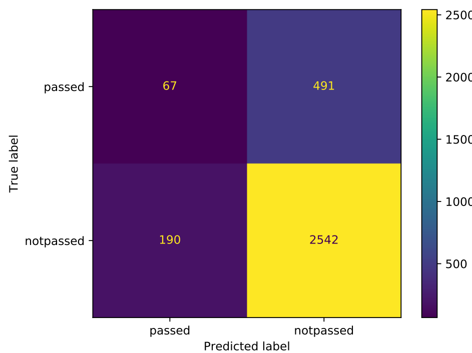
Looking at the confusion matrices in the figure 6.3, it seems that the model based on the *code vectors* can pick up some similarities and differences in order to determine whether or not a certain exercise could result in a successful exam question.



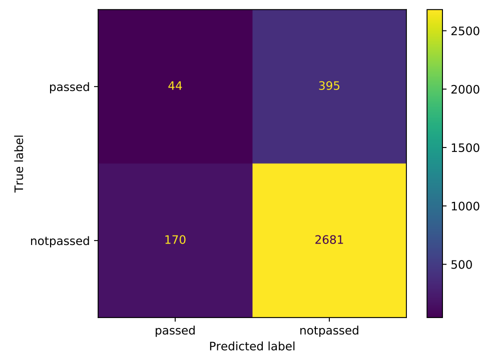
(a) *Exercises on question q1* outcome



(b) *Exercises on question q2* outcome



(c) *Exercises on question q3* outcome



(d) *Exercises on question q6* outcome

Figure 6.3: Confusion Matrices for *Exercises on Exam Questions outcome* using model on *code vectors*

Unlike the previous case, this time we see a large number of elements in the class of *False Positives*. This may be due to the fact that especially in questions *0119_q3* and *0119_q6* the labels are strongly unbalanced towards the class *NotPassed*.

	Precision	**Recall**	**F1-Score**	**Support**
Passed	0,41	0,28	0,34	1227
Not Passed	0,64	0,76	0,7	2063
Accuracy			0,58	3290

Table 6.6: Classification Report for *Exercises on Exam Question q1* outcome using model on *code vectors*

	Precision	**Recall**	**F1-Score**	**Support**
Passed	0,39	0,27	0,32	1136
Not Passed	0,67	0,78	0,72	2154
Accuracy			0,61	3290

Table 6.7: Classification Report for *Exercises on Exam Question q2* outcome using model on *code vectors*

	Precision	**Recall**	**F1-Score**	**Support**
Passed	0,26	0,12	0,16	558
Not Passed	0,84	0,93	0,88	2732
Accuracy			0,79	3290

Table 6.8: Classification Report for *Exercises on Exam Question q3* outcome using model on *code vectors*

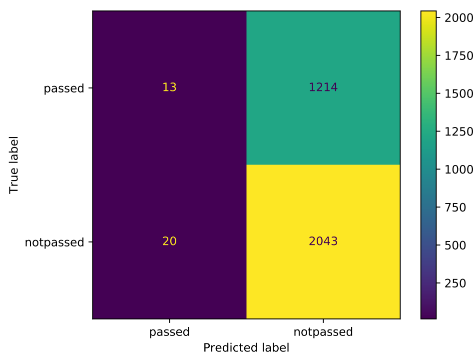
	Precision	**Recall**	**F1-Score**	**Support**
Passed	0,21	0,1	0,13	439
Not Passed	0,87	0,94	0,9	2851
Accuracy			0,83	3290

Table 6.9: Classification Report for *Exercises on Exam Question q6* outcome using model on *code vectors*

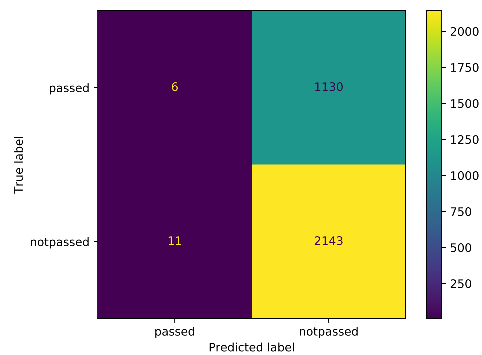
These models tend to have better accuracy percentages (see tables 6.6, 6.7, 6.8, 6.9) than the one shown in the first analysis, but these percentages are also "dangerously" close to the percentage of labels belonging to the negative class.

6.3.3 Analysis on Path Vector

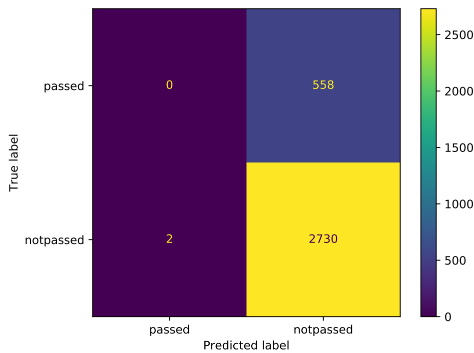
Again, models built on *path vectors* were unable to distinguish examples belonging to the class *Passed* and assigned almost all the code snippets to the predominant class, *NotPassed*.



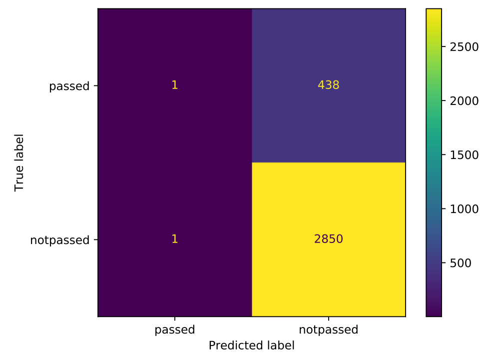
(a) *Exercises on question q1 outcome*



(b) *Exercises on question q2 outcome*



(c) *Exercises on question q3 outcome*



(d) *Exercises on question q6 outcome*

Figure 6.4: Confusion Matrices for *Exercises on Exam Questions outcome* using model on *path vectors*

As the confusion matrices in figure 6.2 show, apart from very rare exceptions for questions *0119_q1* and *0119_q2*, that all the exercises were placed in the class *NotPassed*.

The same result can also be seen from the reports in the tables below (6.10,

6.11, 6.12, 6.13) where the values of *recall* for the *Passed* label are between 0 and 0.01, confirming the poor performance of the models.

	Precision	**Recall**	**F1-Score**	**Support**
Passed	0,41	0,01	0,02	1227
Not Passed	0,63	0,99	0,77	2063
Accuracy			0,63	3290

Table 6.10: Classification Report for *Exercises on Exam Question q1* outcome using model on *path vectors*

	Precision	**Recall**	**F1-Score**	**Support**
Passed	0,35	0,01	0,01	1136
Not Passed	0,65	0,99	0,79	2154
Accuracy			0,65	3290

Table 6.11: Classification Report for *Exercises on Exam Question q2* outcome using model on *path vectors*

	Precision	Recall	F1-Score	Support
Passed	0	0	0	558
Not Passed	0,83	1	0,91	2732
Accuracy			0,83	3290

Table 6.12: Classification Report for *Exercises on Exam Question q3* outcome using model on *path vectors*

	Precision	Recall	F1-Score	Support
Passed	0,5	0	0	439
Not Passed	0,87	1	0,93	2851
Accuracy			0,87	3290

Table 6.13: Classification Report for *Exercises on Exam Question q6* outcome using model on *path vectors*

6.4 Exam Questions Outcome

Given the not so encouraging results obtained with the first two analyses, we have tried to change the focus of our research.

We wanted to evaluate how much *code2vec* is able to extrapolate information from the source code. So we decided to see if it is possible to evaluate an exercise simply by analyzing the source code written by the student, without running and testing it.

6.4.1 Dataset

For this analysis we took into consideration the datasets for each of the four examination questions (*0119_q1*, *0119_q2*, *0119_q3*, *0119_q6*), the ones discussed in chapter 5.3.1.

In each of the four datasets, to every code snippet was given the label *Passed* if that exercise actually scored more than 50% during evaluation performed automatically by INGInious and *NotPassed* otherwise.

In the table 6.14 we show the size of the datasets, the distribution of the labels and the number of paths and tokens. Again as in the preceding analysis, the percentage of *Passed* labels decreases with increasing difficulty of the exam questions, especially for datasets *0119_q3* and *0119_q6*.

Dataset	**Samples**	**Passed**	**Passed (%)**	**Tokens**	**Paths**
0119_q1	606	365	60,2	351	3509
0119_q2	568	339	59,7	419	4110
0119_q3	562	176	31,3	661	5196
0119_q6	470	124	26,4	286	4998

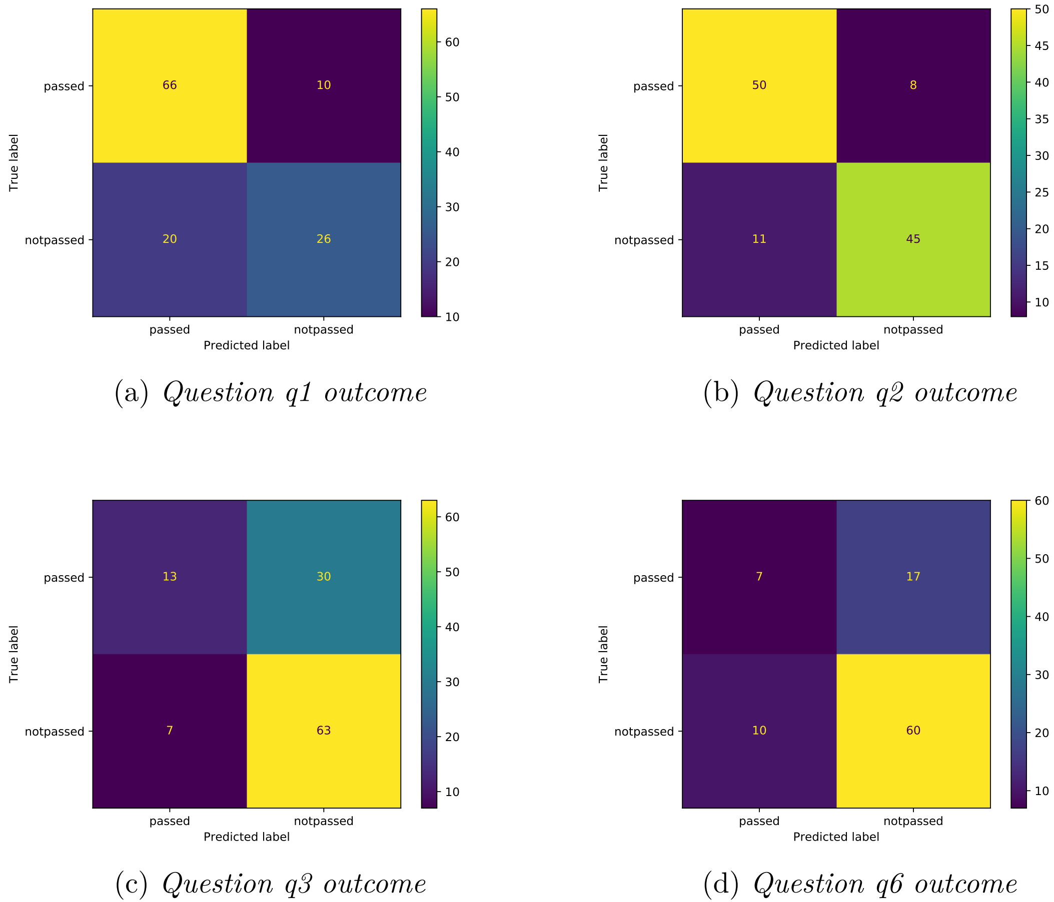
Table 6.14: Exam Questions outcome datasets information

6.4.2 Analysis on Code Vector

The results on *code vectors* seem particularly promising, especially for the first two examination questions.

From the confusion matrices in the figure 6.5 we can see how the models tend to classify quite correctly the outcome of the examination based on the pure evaluation of the vectors extracted by *code2vec*.

In particular, the question *0119_q2* seems to be easily classifiable reaching an accuracy level higher than 80% (Table 6.16).



(a) *Question q1 outcome*

(b) *Question q2 outcome*

(c) *Question q3 outcome*

(d) *Question q6 outcome*

Figure 6.5: Confusion Matrices for *Questions on Questions outcome* using model on *code vectors*

The models built on *0119_q1* and *0119_q3* also show good performance although they tend to be unbalanced towards the classes of *False Positives* and *False Negatives* respectively.

Finally, *0119_q6* tends to classify most entries with the label *NotPassed*, but probably because more than 70% of the students failed to pass this question.

	Precision	**Recall**	**F1-Score**	**Support**
Passed	0,78	0,88	0,83	76
Not Passed	0,75	0,59	0,66	46
Accuracy			0,77	122

Table 6.15: Classification Report for *Question q1* outcome using model on *code vectors*

	Precision	**Recall**	**F1-Score**	**Support**
Passed	0,82	0,86	0,84	58
Not Passed	0,85	0,8	0,83	56
Accuracy			0,83	114

Table 6.16: Classification Report for *Question q2* outcome using model on *code vectors*

	Precision	**Recall**	**F1-Score**	**Support**
Passed	0,65	0,3	0,41	43
Not Passed	0,68	0,9	0,77	70
Accuracy			0,67	113

Table 6.17: Classification Report for *Question q3* outcome using model on *code vectors*

	Precision	**Recall**	**F1-Score**	**Support**
Passed	0,38	0,25	0,3	24
Not Passed	0,77	0,86	0,81	70
Accuracy			0,7	94

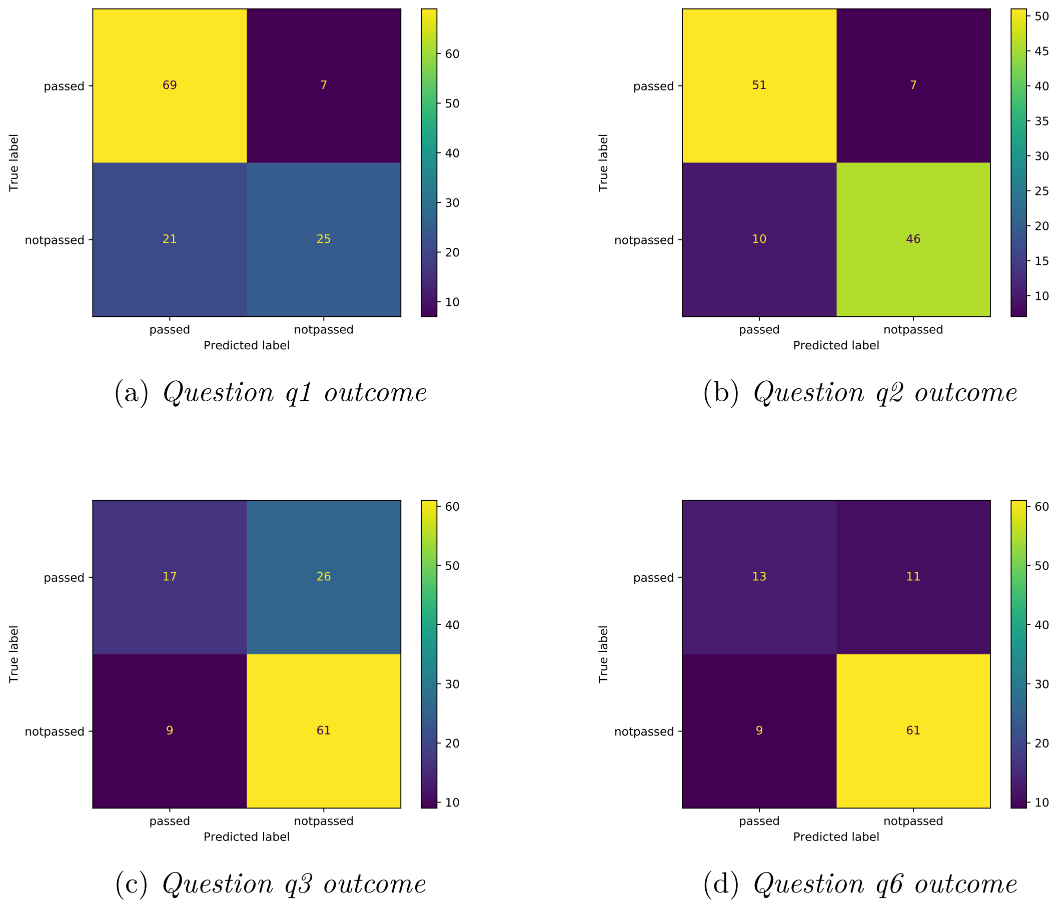
Table 6.18: Classification Report for *Question q6* outcome using model on *code vectors*

As we see from the tables (6.15, 6.16, 6.17, 6.18), the first two models are the best performing with good *precision* and *recall* values in both classes. The other two tend to have worse values, especially of *recall*, for the *Passed* class, but this can be due both to the greater unbalance of the datasets, in fact less than a third of the students managed to pass these questions, and to the fact that the questions are more complex than those proposed in the first exercises. Therefore it should not be easy to find errors specific to the exercise.

6.4.3 Analysis on Path Vector

For this analysis the *path vectors* finally seem to get good performances, even better than those obtained by the *code vectors* just considered.

The confusion matrices in figure 6.6 show very promising results especially for the first two datasets. In fact, it seems that just the presence or absence of certain paths can be a strong indicator to differentiate the exercises performed correctly from the wrong ones.



(a) *Question q1 outcome* (b) *Question q2 outcome*

(c) *Question q3 outcome* (d) *Question q6 outcome*

Figure 6.6: Confusion Matrices for *Questions on Questions outcome* using model on *path vectors*

As we can see from the classification reports (table 6.19, 6.20, 6.21, 6.22) all models achieve accuracy rates above 75% except the one for the question *0119_q3*, which doesn't show a big increase in performance and always tends to have several elements among the *False Negatives*.

	Precision	**Recall**	**F1-Score**	**Support**
Passed	0,77	0,91	0,83	76
Not Passed	0,78	0,54	0,64	46
Accuracy			0,77	122

Table 6.19: Classification Report for *Question q1* outcome using model on *path vectors*

	Precision	**Recall**	**F1-Score**	**Support**
Passed	0,82	0,88	0,85	58
Not Passed	0,87	0,8	0,83	56
Accuracy			0,84	114

Table 6.20: Classification Report for *Question q2* outcome using model on *path vectors*

	Precision	**Recall**	**F1-Score**	**Support**
Passed	0,68	0,35	0,46	43
Not Passed	0,69	0,9	0,78	70
Accuracy			0,69	113

Table 6.21: Classification Report for *Question q3* outcome using model on *path vectors*

	Precision	Recall	F1-Score	Support
Passed	0,62	0,54	0,58	24
Not Passed	0,85	0,89	0,87	70
Accuracy			0,8	94

Table 6.22: Classification Report for *Question q6* outcome using model on *path vectors*

As we said at the beginning the *path vectors* are much more interpretable than the *code vectors* and given the good results obtained in this task we decided to study more in depth the interpretability of these results. This will be discussed in more detail in chapter 7.

6.5 Final Considerations

Our first two analyses were a good first attempt to approach the student assessment task, but they had a big underlying problem: they were not able to assess the student as a whole, but only the single exercise. In this way you cannot keep track of whether a student may have done all the exercises assigned to them during the year or only just a few of them.

In fact, in both analyses each exercise is treated as an independent entity, regardless of which exercise you are evaluating in particular or the student who wrote it. This is a big limitation because it does not allow us to evaluate the student as a whole, as we were able to do in the analysis in chapter 4.

As we will describe in the chapter 8.1 we had some ideas about how this project could be improved and developed further.

The last analysis, the one in which only from the representation of the code we try to understand if it is of sufficient quality to pass this test, has brought encouraging results and in fact it will be further deepened in the next chapter.

Given the results obtained for the examination questions we have tried to apply the same approach to the exercises carried out during the course of the year in order to see if it was possible to find the same kind of information obtained with the examination questions.

However, for the exercises assigned during the year, students have "unlimited" time to do them and consequently tend to try them until they reach the maximum grade. As a result the datasets generated by taking the best submissions of an exercise for each student were extremely unbalanced towards the class *Passed* (over 90% of the submissions taken into consideration belonged to that class) resulting in models that classified each exercise as *Passed*.

A possible solution could be to evaluate all of a student's submissions, in this

way you could evaluate which *path_contexts*, or simply which *paths* led to an improvement in the solution.

Chapter 7

Interpretability of Code2Vec Results

Starting from the encouraging results obtained for the source code analysis of the examination questions discussed in chapter 6.4, we decided to go further into the study of the interpretability of the results to see if certain key paths can determine the success or failure of an examination question.

The chapter is divided into the following sections:

- *Procedure Description* - in which we will discuss the idea and method adopted to carry out this study, describing the use of *attention vector* and the procedure for extracting *paths*.

- *Example* - where we will report an actual example of how this study can be applied to specific exercises, in this case extracted from the question dataset *0119_q1*.

- *Considerations* - where we will briefly describe the results obtained and possible suggestions for improvement.

7.1 Procedure Description

In this section, we will describe the idea and procedure used to evaluate the *paths* that are important for the evaluation of source code.

As we know, the principle behind the *code2vec* model is that not all *paths_contexts* have the same importance to distinguish different methods. Likewise, the assumption behind our idea is that some *paths* may be more important than others in determining whether the code of a specific exercise is written correctly or not.

7.1.1 Attention Vector

As we said in the chapter 5.5, at the end of the whole pipeline consisting of: file extraction, data preprocessing, training of the *code2vec* model and data postprocessing; we get a *JSON* file that contains the information for each snippet of code and, among them, there is a vector that contains the 10 *context_paths* that received the most attention.

Below is an example of the structure of this vector.

```
"attentions": [
    {
        "score": 0.2427450567483902,
        "path": " (LT_EQ)^(comparison)^(comparison)_(NAME)_",
        "token1": "<=",
        "token2": "x"
    },
    {
        "score": 0.1321692168712616,
        "path": "
            (LT_EQ)^(comparison)^(while_stmt)^(while_stmt)_(suite)_(NEWLINE)_",
        "token1": "<=",
        "token2": "_"
    },
    {
        "score": 0.0983961671590805,
        "path": " (NAME)^(comparison)^(comparison)_(LT_EQ)_",
        "token1": "sum",
        "token2": "<="
    },
    ...
],
```

These *attention vectors* are what we used as starting point for our analysis.

7.1.2 Paths Extraction

First of all, we only extracted the *paths* from every *path_context*. We focused only on the *paths* because, in these cases, the model evaluates all the submissions for the same exercise. So we thought that the information on the *path* was sufficient, while by considering the *paths_contexts* we would have kept also the information on the *tokens* which, since they often contain names arbitrarily assigned by the students, could differentiate some of the *paths* of the ASTs that express very similar concepts.

Then we created a dictionary where the key is the string representing the *path* and the values are calculated by counting how many times that *path* appears in the *attention vector* of a method that will be then assigned to the *Passed* class and

doing the same thing for the *NotPassed* class.

From this data structure we created a table and looked for which *paths* had a big discrepancy between the two values, in particular we were interested in finding *paths* that would lead to a negative code classification. In the table 7.1 we can see some of the most frequent *paths* extracted from the dataset of the exercise *0119_q1* along with the number of times they appear in each class.

Id	**Path**	**Passed Prediction**	**Not Passed Prediction**	**Diff**
1	*(NEWLINE)^(simple_stmt)^(suite)^(suite)_(simple_stmt)_(expr_stmt)_(NAME)_*	441	161	280
2	*(NAME)^(power)^(expr_stmt)^(simple_stmt)^(simple_stmt)_(NEWLINE)_*	205	64	141
3	*(NEWLINE)^(suite)^(suite)_(simple_stmt)_(expr_stmt)_(NAME)_*	78	2	76
4	*(DEDENT)^(suite)^(while_stmt)^(suite)^(suite)_(simple_stmt)_(return_stmt)_(arith_expr)_(NAME)_*	73	0	73
5	*(NUMBER)^(expr_stmt)^(simple_stmt)^(suite)^(suite)_(simple_stmt)_(expr_stmt)_(NAME)_*	84	14	70
6	*(GREATER_THAN)^(comparison)^(comparison)_(NAME)_*	7	62	55
7	*(ADD)^(arith_expr)^(arith_expr)_(NUMBER)_*	67	13	54
8	*(NAME)^(comparison)^(comparison)_(GREATER_THAN)_*	14	67	53
9	*(NAME)^(power)^(expr_stmt)^(simple_stmt)^(suite)^(suite)_(simple_stmt)_(expr_stmt)_(NAME)_*	61	9	52
10	*(WHILE)^(while_stmt)^(while_stmt)_(comparison)_(LT_EQ)_*	0	41	41

Table 7.1: Most frequent paths for question *0119_q1* with number of appearances in each class prediction

Once we selected some interesting *paths*, which are those with a strong difference between the two values or possibly present in one of the two classes, we went to look through the individual submissions of the students to evaluate the code and represent on an AST the *paths* that received the most attention.

7.2 Example

We decided to analyze the first of the exercises of the January 2019 exam because the required code is short to write and easy to understand. Below is a trace of it translated in english.

> In mathematics, Faulhaber's formula, named after the German mathematician Johann Faulhaber, expresses the sum of:
>
> $$\sum_{k=1}^{n} k^p = 1^p + 2^p + 3^p + ... + n^p$$
>
> where parameters n and p are positive integers and n is strictly positive. For example, for $n = 6$ and $p = 2$ we obtain the sum
>
> $$1^2 + 2^2 + 3^2 + 4^2 + 5^2 + 6^2 = 91$$
>
> You are asked to write the body of a function *faulhaber_max(x,p)* which returns, for an integer $x > 0$ and a given value of $p \geq 0$, the largest integer $n > 0$ for which the result of Faulhaber's formula is strictly smaller than x ($< x$). For example, *print(faulhaber_max(120,2))* prints the value 6 because
>
> $$1^2 + 2^2 + 3^2 + 4^2 + 5^2 + 6^2 = 91 < 120$$
>
> but
>
> $$1^2 + 2^2 + 3^2 + 4^2 + 5^2 + 6^2 + 7^2 = 140 \geq 120$$

The exercise is very simple and requires few lines of code to be completed.

As we have seen in the table above, some paths tend to appear more often in some classes than others. In our case we are interested in selecting those that often appear in solutions classified as *Not Passed*. In particular from the table 7.1 we can see how the tenth *path* seems to be discriminating to negatively classify an exercise.

Looking through the submissions that had such a *path* we found that of a student who got only 50% as a final grade (score judged insufficient for how we set up the analysis). Below is the source code uploaded by the student.

```python
def faulhaber_max(x,p):
    n=0
    n_p=0
    while n_p<=x:
        n+=1
        n_p+=n**p
    return n-1
```

The exercise done by this student was classified with 99% confidence within the class *Not Passed* by the model *code2vec*.

So we extracted the five *paths* with the highest *attention* values (see table 7.2) and generated the *abstract syntax tree* (figure 7.1).

Id	Path	Attention
1	*(LT_EQ)^(comparison)^(comparison)_(NAME)_*	0,275
2	*(NEWLINE)^(simple_stmt)^(suite)^(suite)_(while_stmt)_(comparison)_(LT_EQ)_*	0,152
3	*(LT_EQ)^(comparison)^(while_stmt)^(while_stmt)_(suite)_(NEWLINE)_*	0,150
4	*(NAME)^(comparison)^(comparison)_(LT_EQ)_*	0,070
5	*(WHILE)^(while_stmt)^(while_stmt)_(comparison)_(LT_EQ)_*	0,041

Table 7.2: Paths with higher attention from student submission

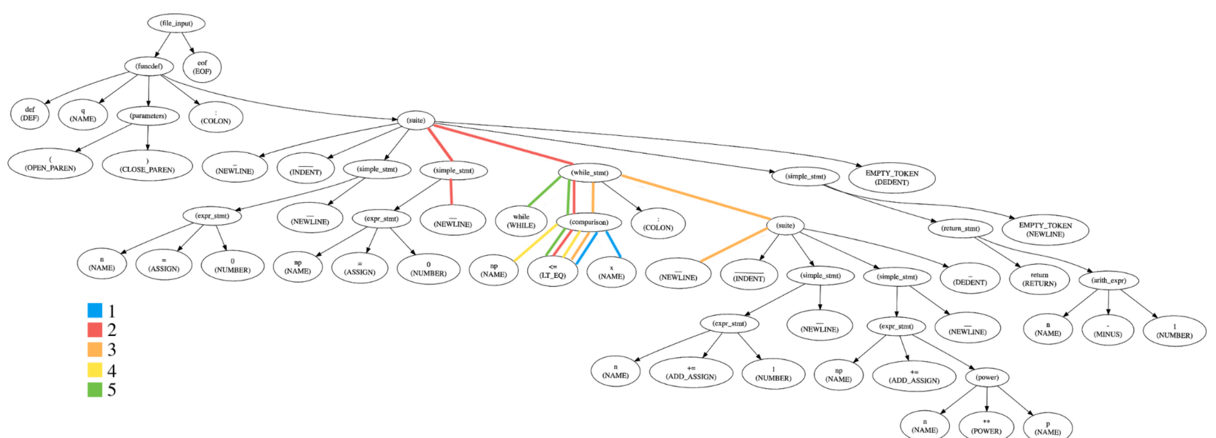


Figure 7.1: AST generated from the student's source code

In the figure 7.2 we see that all the *paths* that received the most attention connect to the node containing '<='. In the exercise text it is explicitly specified that the generated number must be strictly lower than the one in imposed as a limit (p). In fact, going to check on *INGInious*, it is possible to notice that the program does not pass some tests on *corner cases*, that are precisely the cases where the sum of the numbers is equal to the number passed as p parameter.

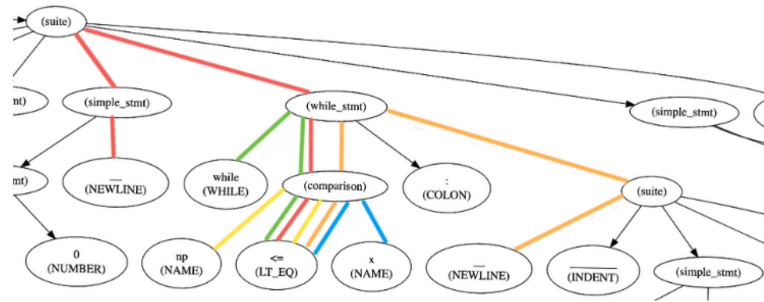


Figure 7.2: Detail from the AST generated from the student's source code

7.3 Considerations

As we have seen in this small study, it is possible to isolate specific *paths* that are able to classify an entire code snippet. Unfortunately with other exercises we were not able to get such clear results.

In general, having richer datasets could have helped to better identify the most recurrent *paths* and thus distinguish the most frequent ones in the different classes.

In addition, more difficult exercises could depend on a larger number of *paths*. In fact, the more complex the code to write is, the greater the number of variables to take into account and so the number of possible errors that can be made. It would be very interesting to further explore this topic with larger datasets and by applying other study techniques, for example by searching sets of *paths* through pattern mining techniques.

Chapter 8

Conclusion

This project represents an exploratory analysis on how to apply the study of the source code in order to draw useful information and make predictions about students' performance.

The first analysis, in which we used qualitative features for the exercises, was used as a starting point to see if it is actually possible to evaluate students based on the work done during the course. The results confirmed expectations with accuracy scores of above 70%.

As far as the source code analysis is concerned, we did not always get the expected results, but there were some interesting insights.

In the first two studies we wanted to test whether the exercises carried out during the year could give useful information to predict the student's exam results.

The models generated, however, did not achieve good performance results and the main reason is the lack of an overview of the student. In fact, the single exercises are not able to give a global idea of how a student develops his own code. Moreover, different exercises have different topics and difficulties and for this reason it would be interesting to divide them into categories.

In the third research we have tried to understand if from the study of the source code only, that is without the need to run it and evaluate its output, it would be possible to judge the good development of it in answering the examination questions. In spite of the small and, in some cases, unbalanced datasets the performance were better than expected and it was decided to go deeper into the subject.

Finally, in our study of the interpretability of the results obtained by *code2vec*, the aim was to understand whether specific *paths* could be indicators of the success of an exercise and possibly of the quality of code writing. The model seems to be able to extract such information and for simple exercises it is easy to interpret these results.

8.1 Limits of the Analysis and Possible Improvements

Among the limits of the analysis we can certainly consider the relatively small number of students and consequently of exercises in our datasets. In fact, as we know, machine learning algorithms obtain better results with greater amounts of data. This has not prevented us from carrying out research, but it could have guaranteed better results in some analyses. It would be interesting to repeat some of the research on larger datasets containing the exercises of the following years in order to be able to evaluate even more students.

8.1.1 Students Evaluation

One problem found in the first two source code studies was certainly the lack of an overall view of all the exercises carried out by the same student. Unfortunately, each student performs a different number of exercises and there is no obvious way to aggregate the source code vectors among them.

One idea might be to implement an extra neural network that tries to summarize the various code vectors into one or use an attention mechanism, similar to what *code2vec* does to aggregate several *path_contexts* to represent a single code snippet. This would lead to loss of useful information about the representations of the individual exercises and further diminish the ability to interpret the results.

Another approach could be to create models for each of the exercises assigned during the year and then aggregate the predictions made by each of them to evaluate the student. It remains to be seen what would be a good way to combine these results and how to manage students who do only some of the exercises.

Finally, a hybrid approach could be considered between analysing the quality of the exercises carried out during the year and studying the source code of the exam questions to predict whether a student will pass the final test. For example, creating a vector in which to enter for each question the probability that it will be sufficient to pass the exam and also add features on the student's performance during the year.

8.1.2 Interpretability of source code

As far as the study of the interpretability of the source code is concerned, we have seen how useful information can be obtained by means of individual *path*. It would be interesting to go further into this kind of analysis to see if, in more complex exercises, certain sets of *paths* can provide information about the quality of the code. One idea would be to use supervised pattern mining techniques to extract groups of *paths* recurring in certain categories of exercises and see if they lead to a positive or negative classification of them.

8.2 Application to Teaching

The deepening of this kind of analysis could be useful to identify, through the study of the source code developed by the students, gaps on specific topics covered during the course, allowing teachers to intervene on the organization of the teaching.

Those proposed in this chapter are just some of the possible future developments for this type of research.

Bibliography

[1] *A Gentle Introduction to the Gradient Boosting Algorithm for Machine Learning.* https://machinelearningmastery.com/gentle-introduction-gradient-boosting-algorithm-machine-learning/. [Online].

[2] Uri Alon et al. «Code2Vec: Learning Distributed Representations of Code». In: *Proc. ACM Program. Lang.* 3.POPL (Jan. 2019), 40:1–40:29. ISSN: 2475-1421. DOI: 10.1145/3290353. URL: http://doi.acm.org/10.1145/3290353.

[3] M. M. Ashenafi, G. Riccardi, and M. Ronchetti. «Predicting students' final exam scores from their course activities». In: *2015 IEEE Frontiers in Education Conference (FIE).* 2015, pp. 1–9.

[4] *Astminer.* https://github.com/JetBrains-Research/astminer. [Online].

[5] *Awesome Machine Learning On Source Code.* https://github.com/src-d/awesome-machine-learning-on-source-code. [Online].

[6] *BSON Wikipedia.* https://en.wikipedia.org/wiki/BSON. [Online].

[7] Zimin Chen and Martin Monperrus. *The CodRep Machine Learning on Source Code Competition.* Tech. rep. 1807.03200. arXiv, 2018. URL: http://arxiv.org/pdf/1807.03200.

[8] *Code2vec.* https://github.com/tech-srl/code2vec. [Online].

[9] *Gradient Boosting Wikipedia.* https://en.wikipedia.org/wiki/Gradient_boosting. [Online].

[10] *Gradient Tree Boosting.* https://scikit-learn.org/stable/modules/ensemble.html#gradient-tree-boosting. [Online].

[11] *GridFS.* https://docs.mongodb.com/manual/core/gridfs/. [Online].

[12] Vladimir Kovalenko et al. «PathMiner: a library for mining of path-based representations of code». In: *Proceedings of the 16th International Conference on Mining Software Repositories.* IEEE Press. 2019, pp. 13–17.

[13] *LINFO1101 - Introduction à la programmation.* https://uclouvain.be/en-cours-2019-linfo1101. [Online].

[14] Tomas Mikolov, Quoc V. Le, and Ilya Sutskever. «Exploiting Similarities among Languages for Machine Translation». In: *CoRR* abs/1309.4168 (2013). arXiv: `1309.4168`. URL: `http://arxiv.org/abs/1309.4168`.

[15] Lili Mou et al. «Convolutional Neural Networks over Tree Structures for Programming Language Processing». In: *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*. AAAI'16. Phoenix, Arizona: AAAI Press, 2016, 1287–1293.

[16] *Pymongo*. `https://pymongo.readthedocs.io/en/stable/`. [Online].

[17] Reuven Rubinstein and William Davidson. *The Cross-Entropy Method for Combinatorial and Continuous Optimization*. 1999.

[18] *Scikit-Learn*. `https://scikit-learn.org/stable/getting_started.html`. [Online].

[19] Amirah Mohamed Shahiri, Wahidah Husain, and Nur'aini Abdul Rashid. «A Review on Predicting Student's Performance Using Data Mining Techniques». In: *Procedia Computer Science* 72 (2015). The Third Information Systems International Conference 2015, pp. 414 –422. ISSN: 1877-0509. DOI: `https://doi.org/10.1016/j.procs.2015.12.157`. URL: `http://www.sciencedirect.com/science/article/pii/S1877050915036182`.

[20] *Syllabus interactif*. `https://syllabus-interactif.info.ucl.ac.be/index/info1-theory`. [Online].

[21] Huihui Wei and Ming Li. «Supervised Deep Features for Software Functional Clone Detection by Exploiting Lexical and Syntactical Information in Source Code». In: *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17*. 2017, pp. 3034–3040. DOI: `10.24963/ijcai.2017/423`. URL: `https://doi.org/10.24963/ijcai.2017/423`.

[22] *What is INGInious?* `https://inginious.readthedocs.io/en/v0.6/teacher_doc/what_is_inginious.html#how-does-inginious-work`. [Online].

[23] M. White et al. «Deep learning code fragments for code clone detection». In: *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2016, pp. 87–98.

[24] J. Zhang et al. «A Novel Neural Source Code Representation Based on Abstract Syntax Tree». In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 2019, pp. 783–794.