UCLouvain

epl

École polytechnique de Louvain

CESReS - Code Embeddings for a Student Recommendation System

When BERT becomes your tutor

Author: **Guillaume Steveny**
Supervisors: **Kim Mens, Siegfried Nijssen**
Readers: **Julien Lienard, Benoît Ronval**
Academic year 2023–2024
Master [120] in Computer Science and Engineering

Abstract

Students need feedback on their work. They use this to get a better overview of their level, strengths and shortcomings. However, personalized feedback about their misconceptions is complex due to a lack of means or resources. Even if some mechanisms have proven to be efficient, like Inginious [1], these require cumbersome and time-consuming work to design specific tests and feedback. This thesis objective was to go over these limits by enabling automatic feedback generation thanks to a machine learning model. To do so, we developed a classification architecture following the latest advances in Natural Language Processing. By using code embeddings, i.e. generated vectors on the students' submissions, our system allows one to detect specific misconceptions occurring in code snippets. To get control of the classes and enable the training of our deep neural network, we developed an approach inspired by DeepBugs [2]. The training instances are mutants of original students' submissions where the injected modifications are representative of a set of 14 misconceptions we selected. Our model obtained f1-score values up to 72.9% when predicting an evaluation dataset of students' mistakes. We also highlighted the limits of our current mutation labelling technique and improvements to be conducted by further works. Finally, the model's program we created can interconnect with a graphical interface and provide basic insights into the model's decisions thanks to the Integrated Gradients computation [3].

Acknowledgments

First, I would like to thank the supervisors of this thesis: Pr. Siegfried Nijssen, Pr. Kim Mens and Julien Lienard for their support, feedback, comments and weekly meetings where I was encouraged and guided all year. Thank you for your involvement in this work throughout the year.

I also want to thank the reader of this thesis, Benoît Ronval, who directly showed his interest in the theme and support me during the poster session.

"Computational resources have been provided by the supercomputing facilities of the Université catholique de Louvain (CISM/UCL) and the Consortium des Équipements de Calcul Intensif en Fédération Wallonie Bruxelles (CÉCI) funded by the Fond de la Recherche Scientifique de Belgique (F.R.S.-FNRS) under convention 2.5020.11 and by the Walloon Region".

Special thanks go to my girlfriend, Lucie Fossion for her unwavering support, presence and comments on my achievements during all year.

A personal thanks goes to my grandmother, Monique Gillard, for her helpful comments on what I should take care of and her everyday encouragement to fulfil this work.

I want to thank Emilie Deprez and Paul-Edouard Bertrand Van Ouytsel for their friendly feedback on my manuscript, even if it was only a draft, but also for their support during the last steps. I will also thank my friend Laurent Carlier for his feedback on the misconceptions I identified from the Python tutoring sessions.

Finally, a last thank goes to every student I helped one day, whether I was a student tutor or not. You initially gave me this thesis idea. I hope someone I will thank in advance will continue this project to help even more students.

Contents

1 Introduction **1**
1.1 Context . . . 2
1.2 Problem . . . 2
1.3 Objective . . . 3
1.4 Approach . . . 3
1.5 Obtained results . . . 3
1.6 Thesis outline . . . 4
1.7 External tools . . . 4

2 Related work **5**
2.1 Automated feedback generation . . . 5
2.2 Mining code patterns . . . 6
2.3 Code embeddings . . . 8
2.4 Related work conclusion . . . 11

3 Background **12**
3.1 Code generated structures and graphs . . . 12
3.1.1 Abstract Syntax Tree (AST) . . . 12
3.1.2 Data Flow Graph (DFG) . . . 14
3.2 Machine learning attention . . . 16
3.3 Bidirectional Encoding Representation for Transformer (BERT) . . . 19
3.3.1 Code embeddings . . . 19
3.3.2 Transformer-encoder & BERT . . . 20
3.3.3 RoBERTa . . . 23
3.4 GraphCodeBERT . . . 23
3.5 Interpretability attribution scores . . . 25
3.6 Loss metrics . . . 26
3.7 Performance metrics . . . 27

4 Preliminary study **29**
4.1 Initial considerations . . . 29
4.2 Code2Vec . . . 30
4.3 Code2Seq . . . 30
4.4 CuBERT . . . 31
4.5 CodeBERT & GraphCodeBERT . . . 31

5 Approach & Solution **32**
5.1 Mutation labelling . . . 32
5.2 Dataset selection . . . 37
5.3 Model architecture . . . 39
5.4 Model implementation . . . 42
5.5 Approach & Solution conclusion . . . 42

6 Experiments **43**
6.1 Model parameters . . . 43
6.1.1 Experimental datasets . . . 44
6.1.2 Parameter choices and baseline . . . 45
6.1.3 Results . . . 47
6.1.4 Conclusion . . . 60
6.2 Model validation and dataset size . . . 61
6.2.1 Manual test-set labelling . . . 62
6.2.2 Selected configurations & test process . . . 62
6.2.3 Results & discussion . . . 63
6.2.4 Conclusion . . . 74

7 Conclusion **76**
7.1 Thesis conclusion . . . 76
7.2 Threats to validity . . . 78
7.3 Identified model limits . . . 79
7.4 Further work . . . 79

Appendix **82**
A RedBaron mutation code example . . . 82
B Designed RedBaron rules . . . 84
C Selected semester exercises . . . 87
D Selected misconceptions . . . 89
E GUI interface . . . 97
F Reference dataset statistics . . . 100
G All results of the “Model parameters” experiment . . . 108
H Additional graphs for the “Model parameters” experiment . . . 109

I Evaluation dataset statistics 118
J Evaluation dataset *baseline* results 119
K Label-wise results of *baseline* on the evaluation dataset 130
L Other datasets statistics 132
M Label-wise results for model limited to 5 epochs on the evaluation dataset 136
N Additional graphs for the "Model validation" experiment 138

Chapter 1

Introduction

Learning a language is one of the most fundamental parts of a human being's education. This learning process relies on continuous and constructive help provided by teachers or tutors. The endless feedback on their work, progression and even errors make the students aware of their level and shortcomings, triggering at the same time the construction of self-esteem and evaluation of their knowledge. Nothing is more normal than making mistakes [4]. Learning builds upon this to avoid these errors when facing the same situation a second time. Without such helpful feedback and comments, students cannot realize that their way of seeing the world needs adjustments nor become aware of the reasons driving this necessary reassessment [5, 6, 7, 8].

At the University, some courses target learning a programming language. Students learn with some autonomy and practice by doing projects and exercises. However, providing the necessary feedback becomes more and more complex for teachers. This observation is principally explained by a lack of means or resources to give a personalized follow-up to everyone.

Nevertheless, an alternative for human feedback is emerging: artificial intelligence. The arrival of language models usable by the general public shook the ways of thinking of a wide range of the population. The impact of GPT-3 (ChatGPT core) [9] in various domains is no longer to be proven [10, 11, 12]. Its availability in 2022 created a complete revolution, and its use is not anymore limited to academic research contexts: public health [13], gaming [14], art [15], but also programming. This last category is the one that will drive this thesis with the aim of asking ourselves the question: "Is it possible that a language model plays a role when learning a programming language?"

1.1 Context

Numerous universities offer training in a programming language. UCLouvain does not depart from this digital schooling practice by integrating, in the first Bachelor year in civil engineering and computer science, an introductory course on the Python language. Students following this course have to execute different tasks to certify their knowledge. Learners should read a syllabus teaching the basis of the language, follow theory courses covering the reference book, participate in exercise sessions monitored by a tutor, and produce, each week, a coding assignment where they solve a more abstract problem thanks to the material taught during the previous weeks. This pedagogy requires more active and involved student participation during the semester.

In this teaching process, two mechanisms allow the students to receive feedback on their work: their tutor and the Inginious platform [1]. The tutor meets them each week and skims their submissions to provide a 10-minute global comment. Inginious is an online teaching assistant that provides instantaneous feedback on coding exercises and projects. This platform allows deploying environments to test student submissions automatically.

1.2 Problem

Sadly, these two learning mechanisms have their flaws. The tutor can only devote a limited amount of time to the students he/she monitors, not allowing to help them as much as he/she would. His/her follow-up is only sporadic and global, not always matching the student's expectations. Inginious, for its part, offers unit tests to check the behaviour or, to a lesser extent, the structure of the student-produced code. Even if possible, writing tests highlighting mistakes or misconceptions is difficult and time-consuming, limiting their deployment on most exercises. Furthermore, to understand the reasons behind such errors, students need to interpret the feedback, which could be complex when starting a Bachelor's degree. Feeling less supported in their learning process even with this continuous form of feedback, a non-negligible part of them stops dealing with the pace of weekly submissions. Others rely on generative AI and online resources to do their work for them, missing the personal research aspects of their learning. Reading the syllabus is put aside, and the student progression seems compromised as the semester progresses.

1.3 Objective

A possible solution to drive the students back to personal and reflexive work would be to integrate inside the online platform mechanisms descended from the last Natural Language Processing (NLP) advances. This work will target the creation of a code classification system based on classical misconceptions and frequent errors of students. Every code, once processed by the model, will obtain a prediction associated with a confidence level, triggering practical, usable and meaningful feedback to the students to make them able to identify their shortcomings and how to remedy these. These comments could be provided as hints to highlight the predicted error sources or as links to the syllabus to incite students to read it more frequently.

1.4 Approach

Most of the architectures recently deployed to generate text summaries [16] or for information retrieval tasks [17] rely on embeddings. Dense vectors of real numbers, these mathematical objects hold properties capturing the semantic essence of a language [18, 19, 20]. When built on words, embeddings offer ways to characterize the closeness of meaning by using Euclidian distance, while ones constructed over entire texts can represent the themes approached in each of them. Systems like ChatGPT [11] and Bard [21], to mention just a few, use these structures in their heart. The BERT architecture [22], usable to obtain them, will be the basis of the present research.

To obtain the classification system, the mutation testing procedure [23], like in software development, served as inspiration. Like in the basis of the DeepBugs system developed by Michael Pradel and Koushik Sen [2], a series of mutations allows constructing training examples containing the misconceptions to be detected.

1.5 Obtained results

This thesis developed two main contributions: a mutation labelling program and a classification neural network model. The first allows one to create labelled mutants with a Python program. Using files containing rule sets, the user could specify which mutations he/she wants to perform on Python files. The second can use the mutants to train itself to detect specific injected misconceptions. Furthermore, it can connect to a GUI application built with Java to perform dynamic predictions.

All the source codes produced for this thesis are stored on a GitHub repository[1]. An external appendix provides a zip archive of this repository.

1.6 Thesis outline

The following chapter [2] will briefly present earlier work related to the subjects tackled in this thesis. This part will first cover automatic feedback generation systems as they are linked to the goal of this thesis model [2.1]. We will then present pattern-mining approaches on source code [2.2] before ending on systems generating and using code embeddings [2.3]. The next chapter [3] will provide the theoretical knowledge required to understand the underlying components of our model. Graph structures for generating our mutants or input for our embedder will start this part [3.1] before covering machine learning aspects [3.2, 3.3, 3.4]. Different score metric computations will end this chapter [3.5, 3.6, 3.7]. The fourth chapter [4] will compare different embedder's choices and motivate our final decision. We will continue with the *Approach* chapter [5] which presents the mutation labelling process [5.1, 5.2] and the model architecture we used [5.3, 5.4]. The sixth chapter [6] will explore the two experiments we defined on our model: "model parameters" and "model validation". The first covers the training task we defined, while the second assesses the model performance for real cases. Finally, the *Conclusion* chapter [7] will end our discussion by explaining the limits of our approach and possible further works.

1.7 External tools

No generative AI tool has been used to produce part of this manuscript text. Only Grammarly, in its free version, has been used to highlight syntax and grammar errors. The content of the bibliography has been generated with the help of Zotero. Regarding the code produced, official tutorials from the libraries we constructed our model with were inspiration sources for the code structures and performed computations. These tutorials are the one from `AllenNLP`[2] and `Captum`[3].

[1] `https://github.com/StevenGuyCap/CESReS-model`
[2] `https://guide.allennlp.org/training-and-prediction#4`
[3] `https://captum.ai/tutorials/Bert_SQUAD_Interpret`

Chapter 2

Related work

In this chapter, we will focus on previous work linked with the studied topics of this thesis. These papers and contributions are worth mentioning because they are at the origin of the constructed system. The purpose here is not to go into every detail of the presented research. Thus, pointers to the resources are provided for further reading.

2.1 Automated feedback generation

Since the system we will develop aims to provide even better feedback to the students, exploring existing automated feedback mechanisms is necessary.

For covering different systems, one first approach is to collect surveys and reviews on the domain. A large number of publications give a detailed analysis of existing tools [24, 25, 26, 27, 28].

In one of the recent works [27], the results highlight that most tools rely on output comparison [29, 30, 31] or unit testing [32, 33, 34]. However, both have their limitations when identifying particular misconceptions. The first approach requires inferring the code behaviour solely from specific outputs. While designing tests for the second could be cumbersome.

This thesis aims to overcome these limitations. By using a machine learning approach, highlighting specific misconceptions would become automatic. The decisions are based directly on source codes rather than on particular outputs. The instructor would not need to carefully design tests or hints for each specific question. Only some verification of what the system is allowed to predict would be required. The system predictions could then complement what unit tests or program compilation could detect as errors.

Our approach also allows us to consider other aspects than output compliance when assessing submissions. Coding is one part, but checking coding styles could be a way to help students structure their thoughts. Students could be able to succeed in the exercises but also get used to bad practices. Some tools based on static analysis [35, 36, 37] provide a way to evaluate how students could improve their coding practice. The developed tool in this thesis will also include a way to provide this kind of feedback.

How students receive the feedback should also be investigated. A concrete approach for students is to receive generated hints rather than the number of failed tests. One way is to classify code based on failed instances [38]. Other platforms are more focused on specific teacher instructions [39]. However, the same limitations occur in these techniques. Instructors need to specify the tools' behaviour, and this additional work could slow down their deployment.

Other systems try to return a sequence of modifications to help the student. These instructions can be generated based on a reference solution and an error model containing correction rules [40]. An alternative approach is to create modification paths following what the students could try to improve their solution [41]. Other works tackled the hint generation problem by relying on abstract solution states [42]. Nonetheless, careful design of such hints is necessary. The students should be able to think about the reasons for their mistakes. A risk could be for the student to follow the hints without understanding them deeply.

In this present thesis, the infrastructure relies on the classification of code snippets. Each predicted class would be associated with an indication to explain the reason for the expected problem. Links to the reference book would be then provided to help the student understand what he/she should correct.

2.2 Mining code patterns

As an alternative to code embeddings, some papers explored how pattern-mining techniques could apply to code snippets. One advantage of these techniques is they can provide, to some extent, interpretable results. Automatically detecting recurrent structures can provide a classification procedure for the code instances. Since these techniques offer an equivalent approach to help automated feedback generation, we will present some of them in this section.

A possible approach relies on frequent subtree mining applied to Abstract Syntax Trees (ASTs). This idea is underlying the works of Mens et al. [43] and Liénard

et al. [44], which are based on previous research from the same team [45, 46]. One of the main goals of these approaches was to detect code smells. To achieve this, the authors used a tree miner algorithm named FREQTALS. After running the tool, a set of common patterns occurring more in erroneous submissions is constructed. Based on these examples, unit tests could be designed to check the presence of these patterns inside the ASTs of new submissions. To limit the number of considered patterns, they specified a set of constraints. However, tuning such constraints is the major limitation these papers encountered. Without careful design choices, a considerable amount of similar and less appropriate patterns could be highlighted by the miner.

Other works relied on itemset or sequential pattern mining [47, 48]. The first aims to help software maintenance by modularizing part of programs containing duplicate instructions. The second presents a theoretical comparison between itemset and sequential mining in the context of function-call usage discovery. Nonetheless, graph-based techniques proved to be more efficient since they can better handle statement reordering or insertion [49]. Leveraging the power of graph representation is the approach of the last presented paper to detect cloned codes. This technique comes with a more expensive search but benefits from better results.

The paper by Balanyi and Ferenc [50] also suffers from design choice limitations. The authors system highlights specific structures in C++ codes to detect design patterns. Discovering such structures aims to simplify software development and avoid code duplicates. For this, the authors developed a new XML-like language to describe the patterns to be mined in the source codes. These descriptions can rely on, for example, inheritance or call delegation. However, the required constraints tuning is again one of the main obstacles to efficient deployment. Too precise pattern descriptions will fail to identify instances containing slight modifications, while more abstract definitions will output more false positives.

The review by Dong et al. [51] gives an additional overview of previous research targeting design pattern mining. Presented techniques include detecting structures from ASTs or XML structures generated from input codes. The authors of this review highlighted different procedures and the way each of these represents the patterns to extract.

The main advantage of all these works is the interpretability associated with the model's predictions. Each highlighted pattern is explainable by the rules used in the tools. Conversely, when using deep network architectures, one could expect to lose part of this information due to their black-box nature [52, 53, 54]. Researchers developed techniques to provide a comparable mechanism for neural networks but with some limitations [3, 55, 56]. In the sixth chapter of this thesis [6], an

experiment will explore some interpretability techniques applied to the system we developed in this thesis.

On the other hand, the main advantage of neural networks would be to relieve users from careful constraints tuning. These techniques are expected to be more robust to slight changes in the identified patterns. Their generalizability should be able to cope with perturbations in the inputs. Furthermore, these would come with a performance gain offered by the deep representations.

2.3 Code embeddings

As explained in the introduction of this thesis, the constructed system will rely on "embeddings". The underlying concepts of these structures are not new, but their latest developments allowed considerable improvement in natural and programming language tasks. These structures have established themselves as an indispensable basis for machine translation, information retrieval or chatbots. The next chapter [3] will explain how the model used in this thesis generates dense vector representations for code snippets. Before we detail this procedure, this section will cover an overview of other systems which rely on such structures.

The DeepBugs model of Pradel and Sen [2] was one of the first models using embeddings to detect bugs inside code snippets. After mutation of the codes to insert synthetic bugs, the embeddings generated by Word2Vec are used to classify the codes. Word2Vec [57] has been developed to generate embeddings on words for a natural language. The system aims to associate close vectors to words co-occurring in the same surrounding context. However, using this model implies working with static embeddings generated for each token. For example, each time the token "for" appears, it is associated with the same vector. However, when detecting bugs, the surrounding context of each keyword is valuable. Relying on such static "embedders" is thus a limitation.

The paper of Karampatsis and Sutton [58] goes one step further than DeepBugs by using the ELMo architecture by Peters et al. [59] on the same task. ELMo (Embeddings from Language Models) overcomes the static limitation of Word2Vec's embeddings by introducing contextual embeddings. The main idea is to use Long Short-Term Memories (LSTMs) to obtain a vectorial representation of each input, which accounts for their actual surrounding context. For example, a "for" placed in a Shakespeare play would have a different consideration than a "for" in a Python program. The underlying principle of an LSTM cell is to consider the token and the state obtained from the previous input to compute the current output. Chaining the cells would allow each token to be considered in relation to all the preceding

inputs. ELMo uses this technique by combining two LSTMs: one for tokens from left to right and one in the other direction. For SCELMo (Source Code ELMo) [58], the system outperformed what DeepBugs achieved.

Other models tried to construct their representation with textual embedders. The work of Efstathiou and Spinellis [60] uses fastText, a successor of Word2Vec working at the sub-words level. However, the results face the same limitations as discussed for Word2Vec. Predicting the next identifier inside a source code does not compare to predicting the next word in a natural language. A model should capture the essence and structure of a program, not only its textual information. Loops, conditional statements and dependencies are components a system should consider to make better predictions.

The idea of Ben-Nun et al. [61] aims to provide a comprehension of program components. The authors decided to work with an Intermediate Representation (IR). The LLVM compiler can generate this representation in a Static Single Assignment (SSA) form. The advantage is to be able to treat different programming languages coherently. The model transforms the SSA into a Contextual Flow Graph (XFG). This specific structure combines the Data-Flow Graph (DFG) and the Control-Flow Graph (CFG). Based on co-occurring identifiers in the IR, a skip-gram model fits the inputs to generate an embedding for each component. These generated vectors can then be used as input for tasks like malicious code analysis or performance prediction. This work was one of the first to provide evidence for their technique to work on multiple languages and tasks of different types. However, the generated representations are static, which leads to the same conclusion as for the previously presented works.

The approach of Code2Vec [62] explores how to predict Java method names based on their body. The authors decided to represent the code snippets as AST paths, as described by Alon et al. [63]. A central system component is the "attention" combining the path embeddings. This computation mechanism was initially presented in a 2014 paper by Bahdanau et al. [64] to address a memorization problem in decoder architectures. The corresponding layer performs a linear combination between the embeddings generated for each context. Each component of the combination is weighted by a score called "attention". All the details about the formulas will be provided in the next chapter [3].

To go further, the same team developed Code2Seq [65], aiming to generate a sequence of output tokens. In this model, the authors performed additional encoding of the AST paths thanks to LSTMs. However, these models are not "large" enough compared to recent advances and deep architectures. Obtainable results are thus, somehow, limited.

Based on the Code2Vec model, Shi et al. [66] constructed a procedure to identify student misconceptions. The authors' initial statement considers feedback crucial but asserts that automatic misconception detection is laborious. By using code embeddings, the researchers tried to group submissions into clusters. They trained Code2Vec to identify if students succeed in pre-defined conditions, like correct procedure definition, usage of a loop or specific methods calls. The model achieved 75.7% accuracy and outperformed naive baselines like Support Vector Machine (SVM) or majority class voting. Their clusters constituted a basis for labelling the student submissions afterwards. However, as stated in their "limitation" section, their work only focused on a particular exercise. Their conclusions could be worse in a global configuration. Furthermore, their approach is only "semi-automatic", which requires humans in the loop to identify misconceptions from the clusters and discard non-meaningful groupings.

Apart from the previously discussed ELMo, a Transformer encoder-decoder [67] architecture allows the generation of contextual embeddings. Such an architecture was initially designed to simplify models using Recurrent Neural Networks (RNNs) or LSTMs. The OpenAI team used this mechanism to create vectorial representations by contrastive pre-training [68]. In their work, they built positive examples composed of a text input and a code entry. With their training process, their system could perform information retrieval tasks. Given a textual input, the model finds the most similar code entry according to a cosine similarity score. The encoder layers act as the embedder to generate a vector for each component of the pairs. However, using their provided API would not be helpful in this thesis. The implied fees and limitations would slow the deployment while not allowing efficient fine-tuning for the studied tasks.

For systems and techniques before 2019, the literature study by Chen and Monperrus [69] provides a list of publicly available models.

Even if ELMo performed well on specific tasks, researchers identified flaws when used for transfer learning. Pre-training an LSTM network for a more general task would not always give optimal fine-tunable weights. Models for NLP tasks should benefit from transfer learning abilities to avoid wasted time and computing resources. This statement is at the origin of the BERT model [22]. Based on the paper by Vaswani et al. [67], the architecture is composed of a stack of Transformer encoder blocks and targets a global pre-training approach. Since its release, numerous variants have appeared, like CamemBERT, RoBERTa, or AlBERT, each targeting a specific task or improvement. In this context, researchers developed CuBERT [70]. In their work, the authors used the BERT-large configuration to automatically detect inserted bugs in a Python code corpus. Apart from a specific tokenizer designed

for the Python language, the training process followed the same description as for the original BERT architecture [22]. We will cover theoretical BERT backgrounds in the *Background* chapter [3.3]. A discussion about using CuBERT for our task will lie in the *Preliminary study* chapter [4].

The Microsoft research team provided alternatives to CuBERT: CodeBERT [71] and GraphCodeBERT [72]. CodeBERT targets Natural Language - Programming Language (NL-PL) tasks over six programming languages. This set of languages is the same for GraphCodeBERT. The CodeBERT researchers' objective was to obtain a pre-trained model fine-tunable for code search or documentation tasks. For this, they used docstrings and code tokens. However, possible improvements to the model included adding the internal code structure to the model inputs. GraphCodeBERT, the successor of CodeBERT, includes these representations. This architecture is the basis of the system developed in this thesis. The following chapter will provide the background required to understand this.

2.4 Related work conclusion

In this chapter, we covered literature related to the problems, objectives and approaches explored in this thesis. We highlighted the additional work needed for output comparison and unit testing to provide comprehensive feedback to students. We also discussed how feedback could be presented to ensure the students deeply understand the reasons for his/her errors. Works relying on pattern-mining have also been covered to give an overview of possible alternatives to our system. However, the need for "constraints" tuning and the impact of slight changes in the patterns have been presented as main limitations. Finally, we explained some other works generating or using embeddings on specific tasks. The benefits of contextual embeddings and some theoretical bases have been covered. The next chapter will explain the details associated with code-generated structures and embedding generation with deep learning architectures.

Chapter 3

Background

This chapter provides the necessary theoretical background for this thesis. For more information on deep learning, since we are not going to detail it here, the paper of LeCun et al. [73] can provide introductory explanations. The following sections will first introduce structures generated from programming languages: Abstract Syntax Trees (ASTs) and Data Flow Graphs (DFGs). The first data structure will be helpful to create the training instances our model needs (see section 5.1). The second will add information to the pre-trained model our architecture will use (see section 3.4). Then, the "self-attention" mechanism, the basis of BERT architectures, will be covered before explaining the pre-trained models considered in this work. We will need these concepts to understand how the vectorial representations are constructed for the code snippets. The model's predictions' interpretability will be covered afterwards. The presented interpretability mechanisms should highlight how our black-box model behaves. Finally, the performance metrics computation will conclude this chapter. These formulas and concepts will be required to evaluate the correctness of our model and compare alternative architecture choices.

3.1 Code generated structures and graphs

Program source code contains various types of information. One way to represent this knowledge is to rely on graph structures. This section will present two data structures: the AST and the DFG. These representations can be used for compilation or behaviour analysis.

3.1.1 Abstract Syntax Tree (AST)

Abstract Syntax Trees, or ASTs, represent a program as a tree structure. These constructions will be needed to introduce the mutation labelling techniques deployed

to create labelled instances for classification. A possible way to obtain such trees is by relying on a Context-Free Grammar (CFG) of the language [74].

A CFG $G = < \Sigma, N, P, S >$ [75] is defined by:

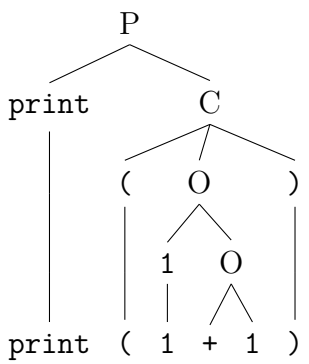
- Σ: the alphabet of terminal symbols
- N: a set of non-terminal symbols
- P: a set of production rules of the form $A \rightarrow \alpha$ with
  - $A \in N$
  - $\alpha \in X^*$ where $X = N \cup \Sigma$ and * is the Kleene star
- S: the start symbol.

The production rules one could apply to produce a particular sequence of terminal symbols can be represented as a tree. This particular structure is called a Syntax Tree. Such trees can also be constructed for programming languages.

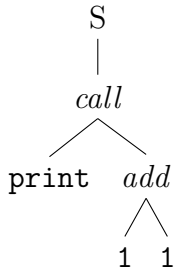
To illustrate the previously discussed aspects, assume a CFG G with:

- $\Sigma = \{\texttt{print},\ \texttt{(},\ \texttt{)},\ \texttt{1},\ \texttt{+}\}$,
- $N = \{P,\ C,\ O\}$,
- $P = \{P \rightarrow \texttt{print}C,\ C \rightarrow \texttt{(}O\texttt{)},\ O \rightarrow \texttt{1}O,\ O \rightarrow \texttt{+1}\}$ and
- $S = P$.

To produce `print(1+1)`, the only possible sequence of production rules is: $P \rightarrow \texttt{print}C \rightarrow \texttt{print(}O\texttt{)} \rightarrow \texttt{print(1}O\texttt{)} \rightarrow \texttt{print(1+1)}$. This sequence could be transformed into the following tree structure.



This tree corresponds to the Concrete Syntax Tree, also called Parse Tree. Dealing with these complex representations can be cumbersome when parsing a programming language before execution. Omitting punctuation marks, separators, and one-to-one transformations could simplify the analysis of the underlying semantics [76]. To do so, one could construct an Abstract Syntax Tree or AST:



Parsers will internally know how to interpret the children of the *call* and *add* nodes. Python relies on these data structures when "interpreting" a user script. The built-in library `ast`[1] can create ASTs and provides utilities to browse and manipulate them.

3.1.2 Data Flow Graph (DFG)

Less focused on the syntactic aspects of a language, the Data Flow Graph (DFG) is another representation of a program. Relieved of some syntactic structures, these graphs capture the origins of the values contained in each variable. Each node represents a variable or a "constant" value like a string or an integer. A directed edge exists between two nodes to indicate a partial origin of the value contained in the targeted node.

To illustrate this graph principle, Listing 1 shows a source code example.

```
def sum_positive_line(x^0):
    bias^1 = 1^2
    weight^3 = 2^4
    if x^5 > 0:
        i^6 = 1^7
        y^8 = 0^9
        while i^10 <= x^11:
            y^12 += weight^13 * i^14 + bias^15
            i^16 += 1^17
    else:
        y^18 = 0^19
    return y^20
```

Listing 1: Python code example from which a DFG will be constructed. The superscript numbers correspond to the node index inside the DFG (see Figure 3.1).

[1] https://docs.python.org/3/library/ast.html

We will follow the definition provided by Guo et al. [72] for GraphCodeBERT. Given a source code $\mathcal{C}$, after parsing it into an AST, we can recover the sequence $V = \{v_1, \ldots, v_n\}$ of values and variables. This sequence can be extracted from the set of terminal nodes in the AST and contains the nodes of our DFG. A variable which is never used or assigned will not be included inside the node sequence. The same holds for constants which are not part of an assignment instruction. Each occurrence of a variable `x` will constitute a unique node v_i inside the directed graph. The DFG $\mathcal{G}(\mathcal{C})$ is the pair (V, E) where E is the set of directed edges. For $i, j \in \{1, \ldots, n\}$, an edge $< v_i, v_j >$ exists ($\in E$) if the value stored in v_j comes from or is computed from the value stored in v_i. If v_i is the left-hand side of an assignment instruction and v_j appears on the right-hand side, there will be a directed edge from v_j to v_i. v_j is the source, v_i the target node. A node v_i can have multiple incoming and outgoing edges, but every node should be part of at least one relationship. Variables that are part of a loop condition have an additional behaviour. The graph needs to reflect the changes that could occur inside the loop. For each variable inside the loop condition which is modified inside the loop body, there is an edge from the updated occurrence towards the other loop occurrences of this variable.

Using the previous definition and the provided code example, one can obtain the following variable and constants occurrence sequence:

```
V = [x, bias, 1, weight, 2, x, i, 1, y, 0, i, x, y, weight, i, bias,
↪  i, 1, y, 0, y]
```

One could notice the absence of the function name (`sum_positive_line`) and a 0 constant (the one at line number 4). The first case corresponds to the fact that the function is never called. The name of this function does not appear in the next instructions; this occurrence is ignored. The second case relates to the fact that this 0 is not part of an assignment and, thus omitted when constructing the graph.

The extracted variables will be interconnected as represented in the directed graph Figure 3.1.

Using this kind of representation could help catching some misconceptions. Students sometimes try to use a variable they never assigned to a value first. Others modify an argument by assigning it a constant value they arbitrarily chose. We expect these cases to be more easily understood by a model leveraging the power of this DFG representation. The *Approach* chapter 5.2 will cover the misconceptions our model aims to detect.

As previously explained in the introduction of this chapter, the DFG structures will be part of the input generation our model will use. More information will be

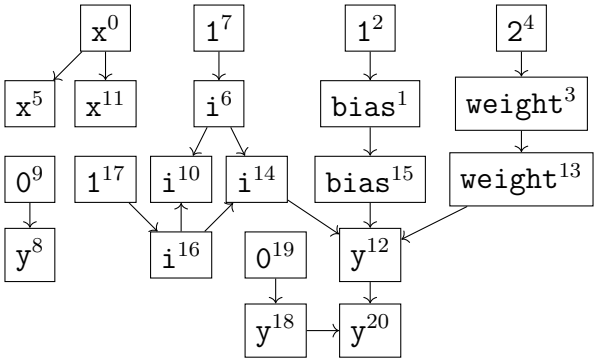


Figure 3.1: Possible DFG construction for the previous example code. The exponent numbers correspond to the position of each occurrence inside the extracted sequence. Each of these constitutes a unique node receiving values from the other occurrences.

provided in section 3.4. As for the ASTs, they will be required to describe our mutation labelling process in the *Approach* chapter [5.1].

3.2 Machine learning attention

The Large Language Models we will explore in this thesis are based on Transformer-encoder blocks. This architecture developed by Vaswani et al. [67] relies on a specific dot-product attention computation. However, the attention mechanism has already been used in other contexts before. One of the first papers to introduce this concept was written by Bahdanau et al. [64]. This section will explain these two computations, but first introduces the underlying abstraction.

As the name *attention* tries to indicate, this mechanism aims to score components to know which are more important than others compared to a particular input. Like a human could do when faced with a complex problem, *attention* will associate high scores with crucial aspects to consider. For example, translating "lie" into "mensonge" (false statement) and not "s'allonger" (recline) in French will require knowing the context of the original word. This kind of information can be found by analysing the surrounding words: the context window. Weighting contributions of words occurring in a context allows the computation to adapt to changes inside this window. This idea constitutes the basis of the *attention* mechanism. The computation output will account for the input values proportionally to their impact on a currently studied component.

We can split the computation into three parts. To simplify, we will consider using the entire context window to compute the scores. However, specifying masks to ignore parts of the input can be considered. We will later come back to this concept,

but will ignore it for now. Figure 3.2 shows a schematic representation of the steps described next [77].

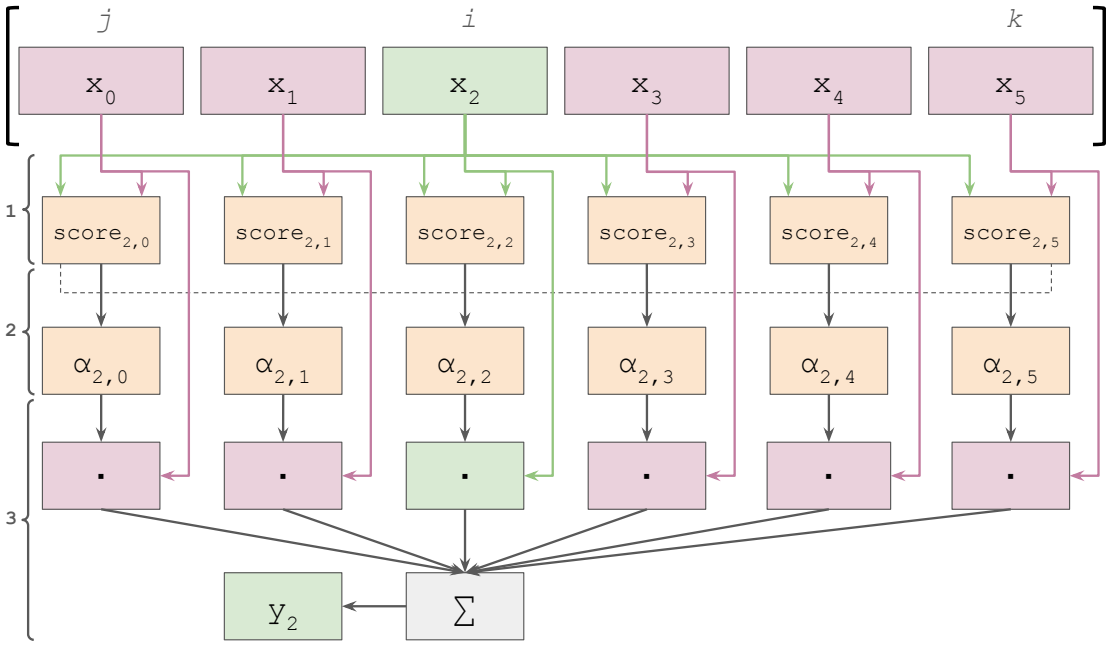


Figure 3.2: Schematic representation of the abstract steps for the attention mechanism.

The steps we highlight here follow the explanations of Bahdanau et al. [64]. For a specific input x_i, giving y_i as output, in a context window from x_j to x_k $(i, j, k \in \mathbb{N} : j \leq i \leq k)$:

1. Computing a correlation score, like a dot product, between the current input and the other values inside the context window:

$$e_{il} = score(x_i, x_l) \; \forall \; l \in \{j, \ldots, k\} \tag{3.1}$$

   This score computation depends on the selected architecture and the input shape of the corresponding layer.

2. Computing the attention value for each previously computed score by normalizing it between 0 and 1:

$$\alpha_{il} = softmax(score(x_i, x_l)) = \frac{exp(score(x_i, x_l))}{\sum_{m=j}^{k} exp(score(x_i, x_m))} \tag{3.2}$$

   where $exp(x) = e^x$ is the exponential function.

3. Performing a linear combination by using the attentions as weights for each of the original values contributions:

$$y_i = \sum_{l=j}^{k} \alpha_{il} \; x_l \tag{3.3}$$

This computation structure is the basic idea behind the attention mechanism, even if some differences can be observed between implementations or models. However, the two papers considered in this section follow this approach.

The computation developed by Bahdanau et al. [64] is called "*additive attention*". The scores e_{ij} are computed by a feedforward network trained jointly with the rest of the architecture. Concretely, we have $e_{ij} = \tanh(w_q \; x_i + w_k \; x_j)$. We can observe two new concepts: w_q and w_k. These are the learned weights vectors from the feedforward network. w_q corresponds to the "*query*" weights (for the inputs for which the output is required), while w_k corresponds to the "*key*" weights (for the values in the window). To leverage parallelization and simplify equations, one can rewrite the complete computation with matrices:

$$\text{AdditiveAttention}(Q, K, V) = \text{softmax}(\tanh(W_Q Q + W_K K))V \tag{3.4}$$

where Q is the query matrix containing each input, K is the key matrix containing the values in the context window and V the value matrix containing the values used for the linear combination.

We presented this computation to introduce the notations and underlying concepts. However, as stated in the paper by Vaswani et al. [67], *additive attention* is much slower and less memory efficient than its "*dot-product attention*" counterpart. In this second form, the linear combination of the previously presented step 3 (equation 3.3) is not performed on the input directly. The x_l values are replaced with a weighted version of these: $v_l = W_V x_l$. The W_V weight matrix will be learned jointly with the model. The W_Q and W_K matrices are still present but have a different role. The score computation becomes:

$$e_{ij} = \frac{(W_Q \; x_i) \cdot (W_K \; x_j)}{\sqrt{d_K}} \tag{3.5}$$

where d_K is the length of the W_K matrix in this formulation.

Once again, following Vaswani et al. [67], we can write a matrix formulation for the complete computation [77]. For this, we can represent our inputs, thus the context window, as a single matrix X of dimension $n \times d$ where n is the number of input values and d the size of each input, considered to be row-vectors. From this input

matrix, we can compute 3 other tensors: $Q = XW_Q$, $K = XW_K$ and $V = XW_V$. The weight matrices W_z are learned jointly with the model and are of dimensions: $d \times d_Q$, $d \times d_K$ and $d \times d_V$.

$$\text{SelfAttention}(Q, K, V) = \text{softmax}\left(\frac{\text{mask}\left(QK^T\right)}{\sqrt{d_K}}\right)V \tag{3.6}$$

One can observe the need for $d_Q = d_K$ for the matrix product QK^T to be consistent. We can also identify a new component: the mask function. Imagine we want to predict the next word to occur in a sentence. To this end, we use attention to highlight which preceding word is the most important to infer the next token. Without masking our context window, here the whole sentence, we will simply have a high score associated with the next word since this token is enough to predict itself. Hence, the SelfAttention has two main operating modes: bidirectional and causal. In the first, every value in the context window is considered; there is no masking. In causal mode, the upper-triangular part of the QK^T matrix is set to $-\infty$, i.e. $\left(QK^T\right)_{ij} = -\infty$ if $i < j$. This masking means that the softmax-normalized scores will be set to 0 (following equation 3.2). There exist other types of masking that could be specified when using the attention layer in practice. We will come back to another attention mask in a future section [3.4].

Other types of attention have also been developed, like MultiHeadAttention aiming to catch different relationships with multiple parallel computations. The goal of this section was to explain the underlying concepts to help understanding the next sections and the model used. We will not go deeper into the details of each existing mechanism. For further information, we recommend reading the paper of Vaswani et al. [67] and the online book "*Speech and Language Processing, 3rd edition (draft)*" by Dan Jurafsky and James H. Martin from Standford University [78].

3.3 Bidirectional Encoding Representation for Transformer (BERT)

3.3.1 Code embeddings

This thesis will use embeddings to catch misconceptions and mistakes inside students' codes. A word embedding is a vectorial representation of a specific word occurrence inside a text [77, 57]. These vectors were introduced to represent semantic relationships between words and be easily usable by computers. This representation technique aims to get embeddings close for semantically close words.

For example, “happy” should be close to “joyful” and far from the embedding generated for “sad”.

We can construct two types of embeddings: static and contextual [59]. In the first case, each word occurrence will get the same value, no matter its surrounding context. The second case will ensure that each generated vector corresponds to the word’s current context. Using “happy” in the sentence “Happy birthday!” will get a different embedding than if used in “Happy New Year!”.

The embedding dimension can also be of two types: sparse or dense [57]. The first case covers embeddings the same size as the learned vocabulary, typically more than ten thousand components. Using these large vectors causes numerous entries to be equal to 0 and makes manipulation by computer more complex. To counter this phenomenon, researchers created dense embeddings. These representations contain fewer entries, typically hundreds, and represent abstract aspects of the words.

The same kind of representation exists for source codes. Contextual dense code embeddings will be the vectorial representations our model will use. The embedder, i.e. the model part responsible for generating the embeddings, will be able to output a global embedding representing a whole code snippet. The following section will explain how this model part creates these.

For further information, the online book “*Speech and Language Processing, 3rd edition (draft)*” we previously mentioned covers more details about the history and construction of word embeddings.

3.3.2 Transformer-encoder & BERT

Section 2.3 already mentioned the Transformer-encoder structure. This architecture is a subpart of the encoder-decoder model developed by Vaswani et al. [67]. Figure 3.3 shows a schematic representation of a Transformer-encoder.

One can observe a self-attention layer corresponding to the mechanism we presented in section 3.2 (Equation 3.6). The classical encoder block uses the Multi-Head variant, where multiple self-attention computations are performed simultaneously to catch different inputs’ aspects.

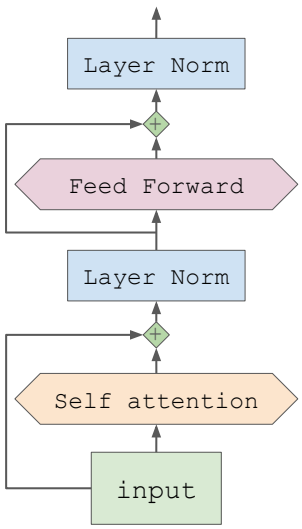


Figure 3.3: Transformer encoder structure.

The original paper presenting this architecture also provided a decoder block structure. However, the embedder our model uses does not rely on this structure. Nonetheless, the meaning of an encoder-decoder architecture can be broader. We can consider an encoder as the sub-part of a model transforming human-readable inputs into an abstract representation the model could use and understand. The decoder block would be the model part translating back the abstract representation into a human-readable output. In our case, the embedder will be our model's encoder. It will generate the embeddings translated afterwards into misconception labels we would associate with the students' submissions.

After consideration of a preliminary exploration (see Chapter 4), the embedder we chose is a variant of BERT. BERT, standing for "Bidirectional Encoder Representation for Transformers", is an architecture developed by Devlin et al. [22]. It contains a feedforward network and a stack of Transformer-encoders. This architecture enables a global pre-training approach. Researchers created two general unsupervised training objectives to obtain pre-trained configuration.

First, we introduce the input representation. When a BERT-like model receives a textual input, a tokenizer splits it into a token sequence. These tokens are part of a vocabulary learned beforehand. The central mechanism to obtain such a vocabulary is to rely on WordPiece [79]. This strategy constructs a set of tokens by looking at the more common character sequences. For example, "`working`" can be split into "`work`" and "`ing`" since each subpart is more frequent than their merging. "`work`" can appear in all its derivations, while "`ing`" appears in the gerund of any verb. Not all the words of a language are part of the vocabulary, but merging different tokens can reconstruct the whole language's word set.

After token sequence generation, the model adds distinctive tokens to the input. `[CLS]` is the classification token. The instance reader of a model prepends this token to the tokenized sequence. The `[SEP]` token marks the separation between sentence pairs in the sequence. A block of tokens between `[SEP]` is called a segment.

The BERT input is called the context window, like for the self-attention mechanism. The original design of the BERT architecture limits this window to 512 tokens. If the tokenized sequence is not large enough to fill the window, `[PAD]` tokens are added to obtain 512 tokens. Each token is then indexed according to the vocabulary: the constructed sequence is the "`token ids`".

There are three other components to give to the BERT model: the "`attention mask`", the "`positional ids`", and the "`segment ids`". The "`attention mask`" relates to the *mask* function we presented in Equation 3.6. This component is a matrix of dimensions 512×512 containing booleans to indicate which pair of

"`token ids`" should be considered at the same time. Setting all the matrix entries to "true" gives a complete bidirectional model.

The "`positional ids`" have been added by Vaswani et al. [67] to represent the position of the "`token ids`". A Transformer-encoder block loses the position information since the self-attention computation ignores the token order. To mitigate this phenomenon, a positional index is associated with each token id. The original proposition by Vaswani et al. [67] relied on cosine functions to attribute these indexes. The GraphCodeBERT model we will describe later [3.4] uses the position of each token in the "`token ids`" sequence. This later section will show an input example.

Finally, "`segment ids`" give the model additional information about the segment in which the token appears. The segment index mechanism works like for the positional indexes: each index is associated with its corresponding token id.

Once all the sequences have been generated, the BERT feedforward network converts each id into a 768-dimensional input embedding. The stack of Transformer-encoders requires transforming the integers to vectors. The generated embeddings are named: "`word embeddings`", "`position embeddings`", and "`segment embeddings`". Then, the model sums the three embeddings for each input token and constructs the 512-length context window to feed the Transformer-encoders.

BERT-base is the lightest configuration the researchers provided. This setting contains 12 encoder blocks, each with 12 self-attention heads. BERT-base will generate a matrix of dimensions 512×768 as output, associating a 768-dimensional contextual dense embedding with each input token. The embedding created for the `[CLS]` token has a specific meaning. It represents the whole input. This vector can be pooled by a "decoder" to generate an output which should contain information about the whole input snippet.

The goal Devlin et al. [22] settled for BERT was to provide a pre-trained architecture one could fine-tune for his/her targeted task. To do so, they defined two pre-training objectives: Masked Language Modelling (MLM) and Next Sentence Prediction (NSP). The first task selects 15% of the tokens at random: 80% of these are replaced with the `[MASK]` token, 10% are replaced by another token and 10% are left untouched. The model should predict, out of the vocabulary, the original tokens of the selected positions. The second objective selects 50% of sentence pairs and replaces the second segment with another in the training corpus. The `[CLS]` embedding will be the input of a feedforward network predicting whether the sentence pair has been modified. Minimizing the sum of cross-entropy losses for the two objectives pre-trains the model.

3.3.3 RoBERTa

"Robustly Optimized BERT pre-training Approach", or RoBERTa, is a variant of the BERT architecture created by Zhuang et al. [80] and the basis of CodeBERT and GraphCodeBERT. RoBERTa-base configuration contains $125M$ parameters, has 12 encoder blocks with 12 self-attention heads and generates 768-dimensional embeddings.

Compared to BERT, RoBERTa has a modified MLM objective. When performing the MLM training for BERT, the model duplicates 10 times the dataset to ensure each instance has different possible masking. However, each mask selection is done once at the start of the process. Each epoch sees the same masked instances. RoBERTa uses dynamic masking. This strategy generates a new MLM mask each time the model receives a sequence as input.

Researchers also found the NLP objective unfavourable to downstream tasks and removed it from RoBERTa. Hence, they deleted the "`segment ids`" from the generated input. They also replaced the WordPiece approach with byte-level BPE (Byte-Pair Encoding). This vocabulary generation process creates tokens by looking at the byte level rather than the Unicode characters. With these changes, RoBERTa managed to outperform BERT on question-answering tasks. We recommend reading the authors' original paper [80] for further information about this model.

3.4 GraphCodeBERT

GraphCodeBERT [72], a successor of CodeBERT [71], targets creating a pre-trained model containing information about six programming languages. The researchers' training dataset contained $2.1M$ text-code pairs and $6.4M$ uni-modal (only code) instances. The programming languages covered by their dataset included Python, Java, Go, JavaScript, PHP and Ruby. Since the model relies on a RoBERTa architecture, it does not contain the "`segment ids`" part. The "`positional ids`" sequence is an ordered sequence of integers representing the token positions. A model input contains three specific segments: the text, the code and the DFG.

The main component the model requires is the code. One could prepend a text segment before the code one. It corresponds to the docstrings extracted from the code inputs. This part can be omitted when the model should focus on the code aspects rather than the pair text-code. In the context of this thesis, we did not use this segment. Section 5.3 will provide the reasons for such a choice.

The DFG is the addition of GraphCodeBERT [72]. Its construction is the same

as we described in section 3.1.2. However, the graph representation should be translated into RoBERTa inputs. To do so, the research team created a strategy to represent the nodes and the edges inside the "`attention mask`": graph-guided masked attention. Given a code $\mathcal{C}$ and a DFG $\mathcal{G}(\mathcal{C}) = (V, E)$ (as defined in section 3.1.2), each node inside V will be represented with an entry inside the "`token ids`". These values will be set to `[PAD]`. The corresponding "`positional ids`" will be set to 0. To avoid confusion with the sequence start and the padding at the end, for which the position ids are set to 1, the code tokens will start the numbering from 2 until reaching the `[SEP]` token. If a node at index i in the "`token ids`" sequence corresponds to a variable occurrence which spans several tokens from j to k inside the same sequence, then:

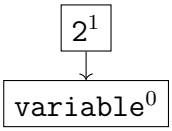
$$mask_{i,j:k} = mask_{j:k,i} = True \tag{3.7}$$

where $mask$ is the "`attention mask`". Furthermore, if there is an edge e_{ij} between the node v_i, stored at the index i' inside "`token ids`", and the node v_j at index j', then:

$$mask_{i',j'} = True \tag{3.8}$$

The following example shows a GraphCodeBERT input with its generated mask.

```
code:
    variable^0 = 2^1

dfg:
```




```
tokens:
    [[CLS], var,   i, able,  =,   2, [SEP], [PAD], [PAD], [PAD]..]

attention mask:
   [[    1,   1,   1,   1,   1,   1,     1,     1,     1,     0..],
    [    0,   1,   1,   1,   1,   1,     0,     1,     0,     0..],
    [    0,   1,   1,   1,   1,   1,     0,     1,     0,     0..],
    [    0,   1,   1,   1,   1,   1,     0,     1,     0,     0..],
    [    0,   1,   1,   1,   1,   1,     0,     0,     0,     0..],
    [    0,   1,   1,   1,   1,   1,     0,     0,     1,     0..],
    [    1,   1,   1,   1,   1,   1,     1,     1,     1,     0..],
    [    0,   1,   1,   1,   0,   0,     0,     0,     1,     0..],
```

```
    [    0,    0,    0,    0,    0,    1,     0,     0,     0,     0..],
    [    0,    0,    0,    0,    0,    0,     0,     0,     0,     0..]]

positional ids:
    [    2,    3,    4,    5,    6,    7,     8,     0,     0,     1..]
```

There is a particularity, principally impactful for the Python language, the input generation GraphCodeBERT misses. There is no specific representation for the line feeds and indentations. For example, the two following code snippets:

```
if 2 > 1:
    print(True)
return True
```

```
if 2 > 1:
    print(True)
    return True
```

will generate the following "`token ids`":

```
[[CLS], if, 2, >, 1, :, print, (, True, ), return, True, [SEP]..]
```

We will come back to this limitation in the *Conclusion* chapter [7] with the *Identified model limits* section [7.3].

Benefiting from the graph representation to outperform CodeBERT, we selected this model as an embedder for our complete architecture. Further details about this choice and the architecture we built will be provided in chapter 4 and section 5.3.

3.5 Interpretability attribution scores

When we will explore the model's predictions in the *Experiments* chapter [6.2], we will want to get insights into the model's decisions. However, as mentioned in the *Related work* chapter [2.2], deep learning models have a black-box nature. To mitigate this effect, we added interpretability mechanisms inside our model's inference process.

These mechanisms compute a score for each token to measure their contributions to the current prediction. We used one specific formula: Integrated Gradients [3]. Other choices, like Layer Conductance [56, 81], are also usable thanks to the `Captum` library [82]. However, during our experiments, we focused on Integrated Gradients for their computation simplicity.

Let's define the model prediction's function for a specific class as $F_c : \mathbb{R}^n \to [0, 1]$. This function corresponds to the output score ("probability") for class c when the model receives a vector of real number of size n. We consider the class-wise

predictions. Let $x \in \mathbb{R}^n$ be the model input, i.e. the input embedding for a token. We set $x' \in \mathbb{R}^n$ as a baseline input, for example the null embedding. Then, the integrated gradients score for the ith dimension of input x will be:

$$IntegratedGradients_i^c(x) = ig_i^c(x) = (x_i - x'_i) \int_{\alpha=0}^{1} \frac{\partial F_c(x' + \alpha(x - x'))}{\partial x_i} d\alpha \quad (3.9)$$

where c is the targeted class, i one dimension of x and $\frac{\partial F_c(x)}{\partial x_i}$ is the gradient of F_c along the ith dimension of x. This definition follows the one provided by Sundararajan et al. [3].

These attribution scores can be computed for each component of each input embedding. Then, the total contribution of the token associated with x is:

$$ig^c(x) = \sum_{i=0}^{n} ig_i^c(x) \quad (3.10)$$

which can be normalized according to the contributions of the other tokens in the input window.

The tokens' contribution can range from -1 to 1. Negative values will correspond to tokens which are not contributing to the class prediction, while positive values should be associated with the tokens impacting the most the output. The **RQ2.5** of the second experiment [6.2] will use the Integrated Gradients attribution scores to determine whether we could use these to interpret the model's decisions.

3.6 Loss metrics

To learn the weights inside our neural network, we need a loss metric. However, as described in the *Approach* chapter [5.2], our model should support multi-label instances. The classical cross-entropy loss, presented in Equation 3.11, is not adapted to this classification type. The equation contains a softmax, which will normalize the logits our model outputs. The loss will push each instance towards one specific label but not all its associated labels.

$$loss_i = -\sum_{c=1}^{C} ln(softmax(x_i, c))y_{i,c} = -\sum_{c=1}^{C} ln\left(\frac{exp(x_{i,c})}{\sum_{k=1}^{C} exp(x_{i,k})}\right) y_{i,c} \quad (3.11)$$

where $loss_i$ is the loss for the instance number i, C the number of classes, ln the natural logarithm, $exp(x) = e^x$ the exponential function, $x_{i,c}$ the logit predicted by the model for instance number i for class c and $y_{i,c}$ the ground truth one-hot encoding of the instance number i for class c.

Instead of using this loss, we will train our model to optimize a multi-label soft-margin loss. Reducing this loss, as if we used cross-entropy, will incite the model to maximize the likelihood of its predictions. The soft-margin term refers to the fact this loss accepts an error margin. The predictions should be as close as possible, but not especially equal, to its ground truth. This loss, presented in Equation 3.12, should push the instances towards all the labels that are associated with them.

$$loss_i = -\frac{1}{C}\sum_{c=1}^{C} ln\left(\frac{1}{1+exp(-x_{i,c})}\right) y_{i,c} + ln\left(\frac{exp(-x_{i,c})}{1+exp(-x_{i,c})}\right)(1-y_{i,c}) \quad (3.12)$$

where $loss_i$, C, ln, exp, $x_{i,c}$ and $y_{i,c}$ have the same meaning as for Equation 3.11.

These formulas come from the official `PyTorch` [83] documentation[2].

3.7 Performance metrics

To measure how the model classifies the students' misconceptions, we will use classical performance metrics. Since our model can target multi-class tasks, we decided to use micro-average variants of these metrics. This section will first present the classical formulas for class-wise computations. We define the subscript c to specify that the different values (TP, FP, ...) are computed over class c out of the C possible classes.

- The precision is the proportion of hits (TP) among the predicted positive (PP) instances.

$$precision_c = \frac{TP_c}{PP_c} \quad (3.13)$$

- The recall, or True Positive Rate (TPR) is the proportion of hits among the positive instances (P).

$$recall_c = TPR_c = \frac{TP_c}{P_c} \quad (3.14)$$

- The False Positive Rate (FPR) is the proportion of false alarms (FP) among the negative instances (N).

$$FPR_c = \frac{FP_c}{N_c} \quad (3.15)$$

[2] `https://pytorch.org/docs/stable/generated/torch.nn.CrossEntropyLoss.html` and `https://pytorch.org/docs/stable/generated/torch.nn.MultiLabelSoftMarginLoss.html`

- The F1-score is a metric combining precision and recall.

$$f1score_c = \frac{2 * precision_c * recall_c}{precision_c + recall_c} \tag{3.16}$$

In the *Experiments* chapter [6], we will use ROC curves to compare different models. ROC curve stands for "Receiver Operating Characteristic curve" and represents the variation of TPR compared to the FPR when changing the prediction threshold. This value corresponds to the limit for which output scores above it are considered predicted, and lower are considered not predicted. This curves should be the closest possible to a square shape, meaning few FP for a high number of TP. To compare curves, one could compute their Area Under Curve (AUC) scores. The closer to 1 this value is, the more the model should be performing.

However, since our model targets multi-label tasks (see section 5.2), we should compute average metrics accounting for the different classes simultaneously. The average over each class, a.k.a. macro-average, would mislead the results since we could have unbalanced datasets (see section 5.2 and Appendix F). The principal alternative is to compute the micro-average of each score. This technique computes the confusion matrix (TP, PP, ...) ignoring the classes. The following formulas provide the alternative computations.

$$precision = \frac{\sum_{c=1}^{C} TP_c}{\sum_{c=1}^{C} PP_c}; \quad recall = \frac{\sum_{c=1}^{C} TP_c}{\sum_{c=1}^{C} P_c}; \quad FPR = \frac{\sum_{c=1}^{C} FP_c}{\sum_{c=1}^{C} N_c}$$

Chapter 4

Preliminary study

Before constructing our model, we need to know which embedder could better suit our needs. Multiple choices have previously been mentioned: Code2Vec [62], Code2Seq [65], CuBERT [70], CodeBERT [71] and GraphCodeBERT [72]. We already explained that we opted for GraphCodeBERT, but the reasons for this choice have not been motivated yet. This chapter will explain a preliminary experiment we performed on each system to drive our choice.

4.1 Initial considerations

The five models explored in this chapter were selected for having their resources available online. They also represent different technologies, techniques and configurations used by different researchers.

Code2Vec [62] and Code2Seq [65] have been developed by the same team. AST paths are converted into dense vectors and then combined using an attention mechanism. The second model outperformed Code2Vec and went further by enabling docstrings generation. The `tech-srl` GitHub account[1] contains the repositories for these two systems.

CuBERT [70] relies on BERT to generate embeddings. The model was trained to automatically detect bugs in open-source code. The authors' GitHub repository[2] provides information about their training procedure and datasets.

Finally, CodeBERT [71] and GraphCodeBERT [72] have been constructed by Microsoft's research teams. Easily usable thanks to the `HuggingFace` library[84],

[1] `https://github.com/tech-srl`

[2] `https://github.com/google-research/google-research/tree/master/cubert`

these models can be directly used to construct systems based on embeddings. The authors' GitHub page[3] contains all the source code for these models.

To compare these architectures, we took a single exam question with a nearly equal number of successful and failed submissions (264 / 244). The model was then trained to predict if the student succeeded or not. This task is complex since it requires understanding the actual code properties and inferring the expected behaviour only from the successful instances.

4.2 Code2Vec

This model seemed interesting as a starting point. Sufficiently small to be trained on a regular computer, it is possible to adapt it for our current task. However, the model was not initially intended for Python, and we used an unofficial implementation from Kirili4ik[4].

Some observations made us think this version contains bugs. First, the developer specified it only works for the `Keras` version, but using this parameter crashes the training script. We need to use the `TensorFlow` version to avoid this behaviour. Furthermore, after training, we observed the displayed precision, recall and f1 scores always equal to each other. Finally, the system only obtained 50% precision (and thus recall) on the validation set and only 20% on the test set.

All these facts combined convinced us not to pursue this approach.

4.3 Code2Seq

Since this model was a successor of Code2Vec, we thought it would correct its previous problems. Furthermore, this model has the advantage of being light and trainable without high-performing GPUs or TPUs. To check its behaviour, we used the same dataset as for Code2Vec. Unfortunately, the model achieved a 0% accuracy score on the test set. We explored possible reasons for this behaviour and found the predictions to be, as we could have been expecting, a sequence of tokens. However, for our task, we should only have one associated label. Predicting the sequence "`failed failed success success`" does not give helpful information.

A possible solution could have been to modify the Code2Seq source code to generate a single word. The last part of the model, acting like a decoder here, would have

[3] `https://github.com/microsoft/CodeBERT`
[4] `https://github.com/Kirili4ik/code2vec`

been trained to output the class prediction. However, as discussed in the Related work chapter [2.3], this architecture is not as robust as we may want. The initial embedding generation relies on dense layers and LSTMs, which showed limited transfer learning power. Self-attention and transformers have recently shown improved performance compared to these mechanisms. Consequently, we decided not to continue with the Code2Seq model.

4.4 CuBERT

CuBERT was the first BERT alternative we considered. It uses BERT-large, another BERT configuration with $340M$ parameters. We modified one of the tasks the model targeted to use our instances for classification. Unfortunately, the personal computers we had were not powerful enough. We needed to use the "Consortium des Équipements de Calcul Intensif" (CÉCI) infrastructure. However, the clusters required using pre-compiled modules incompatible with CuBERT. Hence, we adapted the model thanks to a script from Google[5]. Unfortunately, the model predicted "success" for each instance, leading to a 51% accuracy score.

Due to its results, we decided not to continue with CuBERT. After an extensive set of experiments, we concluded this model not to be optimal for our task. CuBERT requires to hardcode the labels inside the Python program. Adding cases or removing them would have required to modify the source code each time. Furthermore, the model deployment requires too many resources. Even for inference, the program requires the CÉCI infrastructure, which limits the integration on a code submission's platforms like Inginious. Finally, we explored CodeBERT and GraphCodeBERT in parallel with CuBERT, and these obtained better results.

4.5 CodeBERT & GraphCodeBERT

These models' advantage is that researchers provided pre-trained versions of these on the `HuggingFace` platform[6]. We wrote a simple Python program which allowed us to use these models. Since RoBERTa-base is the basis of these models, the program can use lighter devices for training and deployment. When checking their behaviour, the models achieved achieved 85% precision and 70% recall on the test set of the task studied in this chapter. We selected GraphCodeBERT as our final embedder since it includes the DFG representation that should add information to the model's inputs.

[5] `https://www.tensorflow.org/guide/migrate?hl=en`
[6] `https://huggingface.co/models`

Chapter 5

Approach & Solution

Now we have covered the theoretical components needed to construct our model and the possible embedding generation models, we need to explain how we combined these inside our classification model. We will first cover a strategy we developed, inspired by the DeepBugs system [2], to create our training instances. This chapter will then provide insights about the datasets we created and the misconceptions our system should target. Finally, we will present the architecture we built to predict code instances. Implementation details will be briefly exposed at the end of this chapter.

5.1 Mutation labelling

One of the main problems we faced when constructing the classification model was to have enough labelled instances. When looking at the datasets used by CuBERT or CodeBERT, even if pre-trained, these models require thousands of fine-tuning instances. The introductory course we are studying is only in its sixth year. Even if we combined all the final failed submissions the teaching team collected at each exam, we would not be able to compete with these datasets.

Furthermore, even if we had enough snippets, these should be labelled to start the supervised learning process. After running the unit tests on the submissions, a simple grade from 0% to 100% is associated with each input. This "labelling" is not sufficient for the objective of this thesis. We want the system to detect the reasons for the errors and misconceptions of the students, not simply infer the grade for a specific input. The system should provide feedback which complements the unit test outputs.

A possible solution could be to manually inspect each "failed" submission and associate it with a label. However, doing this for such a large number of instances

would be inconceivable. Processing more than ten thousand snippets would have constituted a considerable part of this thesis work. Furthermore, to ensure good results, our labelling procedure should be coherent from the start to the end. Our system should be able to internally "understand" our misconception classes. Instances within a class should share common aspects with the others to help the model classify them. If the reasons making us label an instance change during the process, this would disrupt these shared components. This coherence loss would impact the final model performance. Finally, some code errors we would be able to identify inside a specific submission could not suit other instances. Some problems refer to student misunderstanding of the question statements, others to specific structures only used by the student for this submission. How the "labelling expert" would interpret each of these aspects could introduce bias. Delving inside each submission would not make the approach global enough.

Buggy codes are not limited to student submissions; we can also find some online. We could solve the problem of our small number of unlabelled failed submissions by retrieving our instances from online sources. Nonetheless, finding such examples is laborious and requires a careful procedure. Some projects in which these bugs occur could be complex and done with structures used by more experienced programmers. For example, a bug in the `Pytorch` library working with tensors is expected to be, somehow, conceptually far from simpler students' misconceptions, like confusing a print and a return statement. Even if bugs exist inside open-source projects, finding good ones for our task is tedious and would require a lot of work to filter and label them.

To summarize, we need to collect a large set of instances representative of actual students' misconceptions that are not too specific. These should cover the same context as exam questions, i.e. not using too many unseen concepts. The process to obtain these examples should be coherent and, ideally, automatic to avoid cumbersome manual inspection.

The idea explained in the previously presented DeepBugs system [2] could solve this problem. In their paper, the authors decided to inject synthetic bugs inside their instances and then train the model to detect these. By defining the bugs we want to include inside our dataset, we could benefit from an automatic labelling procedure. Each label will be coherently associated with the instances, and we keep control of the misconceptions we want to predict. The underlying hypothesis is that injecting a bug inside a code supposed correct will create an instance which could be considered as incorrect.

The synthetic insertions should be, nonetheless, carefully designed to represent students' "errors". Being too specific will miss errors that could be manually

grouped, and being too general would misclassify correct snippets. By using this procedure, we expect the system to "understand" the injected misconceptions and be able to detect them on real submissions. If our "mutations" are well representative, each time the model would see a similar structure, it could infer the correct label. We then hypothesize our model to understand the meaning of the "bug" rather than looking for particular synthetic injections. The mutated instructions are local, detecting them would require a global comprehension of the code components. For example, replacing a print with a return is a misconception, while having an additional print instruction inside the correct submission should not be considered. The following section [5.2] will come back to our designed classes.

This reasoning inverts some mining-pattern techniques we have seen in the second chapter. In these works, the codes are unlabelled, and the authors want to find recurrent structures in them. Here, we construct specific structures and want a system that automatically detects similar components. The post-processing needed to interpret the patterns becomes a pre-processing to design the injected bugs.

The mutation labelling technique we want to develop should not be confused with the mutation testing procedure used in software development. This other technique provides a way to check the test coverage of a program. By inserting synthetic components inside the source code, the technique aims to find which tests are robust to these modifications. Here, we need the mutated instances to correspond to possible student submissions. The structure and semantics of the original codes should not be modified outside the inserted components. Furthermore, we can not just rely on randomly replacing part of the codes. Each modification should be associated with the corresponding label in a structured and rigorous manner.

We developed a Python program to execute our mutation labelling procedure. The user can specify a set of mutation rules inside a configuration file. The program parses these rules and goes over the original instances to modify them as desired. However, we need to provide a way for the user to define the rules. These instructions need to be easily created, give freedom to the user in their definition and offer extensive modification power. We found two possible approaches: `Comby` and `RedBaron`.

`Comby`[1] is a tool to perform modifications to source codes. It allows one to specify matching rules used for replacement. Its advantage relies on its ability to specify "holes" containing part of the match. We decided to start with this tool since the user would be able to write the match and replace patterns inside the rules.

[1] `https://comby.dev/`

We will take an example to illustrate what we can construct. By specifying the rule `return :[value] => print(:[value])`, our mutation program will replace all the matches with the right-hand side. The `:[value]` component refers to the content matched using the left-hand side pattern.

```
def is_a_greater_than_b(a,b):
    if a > b:
        return a
    else:
        return b

```

(a) Original code.

```
def is_a_greater_than_b(a,b):
    if a > b:
        print(a)
    else:
        print(b)
```

(b) Mutated code.

Listing 5.1: Mutation example of the `return :[value] => print(:[value])` rule using `Comby`.

However, using this tool is not as powerful as we wanted. The developers designed it to work on different languages like C, Java or Lisp. But these languages have specific ways to indicate statements' blocks, like using `{}` and `()`. Python does not follow this rule, blocks are based on the indentation levels. This means that if a user wants to use a hole for the complete "else-block" in the previous example, he/she needs to either know the number of lines or write complex patterns to take the tabs into account. Otherwise, if statements followed this block, they would also have been matched.

Furthermore, extreme care is necessary to avoid matching unexpected statements. For example, if we want to transform all the `if`s in `while`s, writing the rule `if => while` is not correct. Every `elif` keywords will also match and generate `elwhile`. Conversely, some rules could not match all the possible student structures. If we take our previous rule `return :[value] => print(:[value])`, we can see that, if a student uses `return(a)`, the left-hand side will not match.

Even if this tool offers freedom to specify our mutation rules, the limits it shows and the care we need to define rules made us look for another tool. Instead of mutating the textual input, we could rely on the AST to modify the source code. However, as shown in the third chapter [3.1.1], ASTs contain information losses. Aspects like spacing, comments or parentheses are removed to provide a cleaner representation of the inputs. Reconstructing the code from the AST after mutation will not be able to rewrite these components, possibly creating bias for the model to find non-mutated instances. After searching to solve this problem, we found

RedBaron[2]. This Python library relies on Baron, another module for creating Full Syntax Trees (FSTs). Their particularity is that they are lossless and represent the input code with abstract nodes rather than "grammar" nodes like a Concrete Syntax Tree. Transforming back the FST to the original code gives an output strictly equivalent to the initial input. RedBaron provides all the utility functions to manipulate these structures efficiently. Inside such trees, we can search for specific nodes like integer values, variables or associative parentheses and check their properties.

Thanks to this library, we created 41 rules able to mutate student submissions. For example, the rule remove_assign_while rewrites the source code to remove, inside the associated statement block, all updates to variables used in the while condition.

```
def sum_up_to(n):
    out = 0
    i = 0
    tmp = 0
    while i < n and out < n**2:
        out += i
        print(out)
        i = i + 1
        tmp += out
    return out
```

(a) Original code.

```
def sum_up_to(n):
    out = 0
    i = 0
    tmp = 0
    while i < n and out < n**2:
        print(out)
        tmp += out
    return out
```

(b) Mutated code.

Listing 5.2: Mutation example of the remove_assign_while rule using RedBaron.

As we can see, this kind of rule is more powerful than what we managed to write with Comby. Modifications can span multiple lines, ignore irrelevant statements, get information to infer what it needs and reconstruct the code with the same format. However, this power comes with a negative aspect: these rules must be written as concrete Python code. We can not let the user write an abstract rule to mutate the inputs as our Comby approach offered. For example, for the previously shown rule, we need more than 10 Python instructions to perform the mutation (see Appendix A). We have the power but not the freedom to easily write simple rule.

Our final solution combines the best of both worlds. The configuration files can contain a mix of mutation rules. Either the user uses $ to specify a Comby rule or

[2] https://github.com/PyCQA/redbaron

£ for a `RedBaron` rule. The second type of rule can be selected from our list of 41 predefined mutations. For more information about these, Appendix B provides the rule list and a link to the descriptions we wrote on the GitHub repository associated with this thesis.

5.2 Dataset selection

Now we have defined our mutation labelling procedure, we need to select the instances to mutate. Simply taking snippets from open-source projects online would not be helpful for our task. As explained before, such code snippets are expected to be more complex and not represent structures and instructions students could use in their submissions. The context of these codes could be too far from more basic student productions. Our model could struggle to infer the correct labels and generalize poorly for our feedback objective.

One possible solution could be to take the "success" submissions gathered from previous exam sessions. Here, the quantity problem is no more a limitation since we are creating new instances by mutating the original codes. From a limited number of unlabelled examples, we can obtain a large set of mutants corresponding to specifically chosen and designed "errors". The representativeness of each class is less problematic since we are no longer dependent on actual erroneous student submissions.

One limit we could face is to ensure having mutable instances. We are not sure that every mutation could apply to each snippet. For example, replacing the condition in a while statement only applies if the submission contains a while loop. We could observe class imbalance if the mutations are too restrictive. The selected instances should accept a large range of our mutation rules. Furthermore, our dataset should avoid submissions that are too long. Each of our explored embedders contains a token limit. In our case, we decided to truncate the input to its maximal length. Therefore, if a mutation lies outside our "input" window, the model would not have the right information for its classification. Discussion about this choice will be provided in the next section, cf. section 5.3]. The submissions should be small enough to fit inside the embedder's window but large enough to contain multiple possible mutations.

Unfortunately, exam submissions are more subject to miss these two aspects. Sometimes, maybe due to stress or question complexity, students write long and redundant code, not satisfying our second criterion. Secondly, the questions are designed to be specific to some concepts, like manipulating files or dictionary creation. The contexts behind each question are independent of each other. A

classical exam structure is composed of one question about loops, one about lists, one about dictionaries, one about files and one or two about objects. Such a "isolation" could be unfavourable for our model. The goal is to provide feedback globally, not specifically linked to a particular context or exercise. The system should be usable during the semester, not only at the exam.

Based on this observation, we decided to use submissions to semester exercises as our original instances to mutate. This choice allows us to have more flexibility. Our dataset can benefit from a larger set of questions and contexts. The model is expected to perform better since it would be able to see recurrent structures used in different situations. More students also succeeded in these questions, maybe due to more available external resources compared to the exam. This gives us more available source codes for mutation. These semester exercises are designed to cover all the course materials rather than specific aspects. Using these submissions as original instances would allow the model to get a global overview of each part of the course. Finally, by regrouping multiple questions, we can obtain more examples to mutate, hence more instances to train on.

The underlying hypothesis behind using successful submissions expects mutated codes to become erroneous ones. This observation could not always hold in practice. For example, removing a `return` could happen to be located at an unreachable instruction, not changing the semantics of the input. Caution when designing our injected bugs is required to avoid facing such cases. However, if our mutation rules are well-designed, since we can select a large set of submissions we could expect useless modifications to be quite uncommon.

On the 28th of December, for the first semester of the 2023–2024 academic year, we selected 23 exercises. We decided not to include questions related to objects since they are often longer and more difficult to solve. Appendix C contains the names of each selected question. Out of these exercises, we obtained 13804 successful submissions. This selection kept the last successful submission of each student for the considered exercises. Taking the whole set of success would have created duplicate instances inside the training set.

From these sampled submissions, we can use our mutation labelling program to generate our training sets. The multi-class context of our process seems clear since we should cover multiple misconceptions within a single model. However, we should question whether we are classifying single-label or multi-label instances. Considering students' mistakes, one could conclude the need for multi-label classification. Indeed, students are not programmed; they make mistakes without noticing them and without limiting these to one per submission. Thus, our model should learn that multiple labels are associated with the same source code. However, our mutation

labelling program's initial design did not handle successive mutations on the same input. After a first modification to allow multi-label rules, we extended the program to accept previously generated mutants as input.

Creating multi-label instances relies on mutating previously generated mutants. This process allows injecting multiple misconceptions inside an original source code. But, we need to mind two aspects: reverting rules and duplicate creation. The first situation refers to mutations which would "correct" a previously applied rule. For example, replacing an `if` with a `while` and then replacing the same `while` with its original keyword. To avoid such a case, we created two rule design guidelines. First, each rule set (the set of rules associated with the same label) should not contain rules inverting the behaviour of any other rule set. Secondly, the mutation program will select a rule set if the mutant's labels do not overlap with the ones of the considered rule set.

The second aspect to consider refers to duplicated generated instances. To limit this phenomenon, we store the original file name with the mutant. When filtering the instances after generation, an additional program can remove duplicate instances based on the labels and file names.

Now we have our multi-labelling mutation procedure, we need to create the mutations to inject. After considering the selected exercise contexts and some research [85], we designed 15 rule sets covering 14 misconceptions or mistakes. Appendix D provides the class list with illustrative examples. These labels include using print instead of return (`print_return`), having incoherent loops (`bad_loop`) or forgetting to close a file (`bad_file`). For the complete list, please refer to the related appendix. This discussion will also cover hypotheses and justify the designed classes.

5.3 Model architecture

From the discussion we had in the previous chapter [4], we already know that GraphCodeBERT [72] is an interesting embedder choice. But generating embeddings is not the only part our model needs: the original submissions should be converted into interpretable inputs for the embedder, the `[CLS]` embedding pooled, and finally, this selected vector transformed into scores for each class. Each point represents a sub-part of our model, each with its own parameters. Figure 5.3 gives a schematic representation of our architecture.

The model provides a classification architecture similar to models proposed for sentiment analysis tasks. The instances are transformed into three parts: the

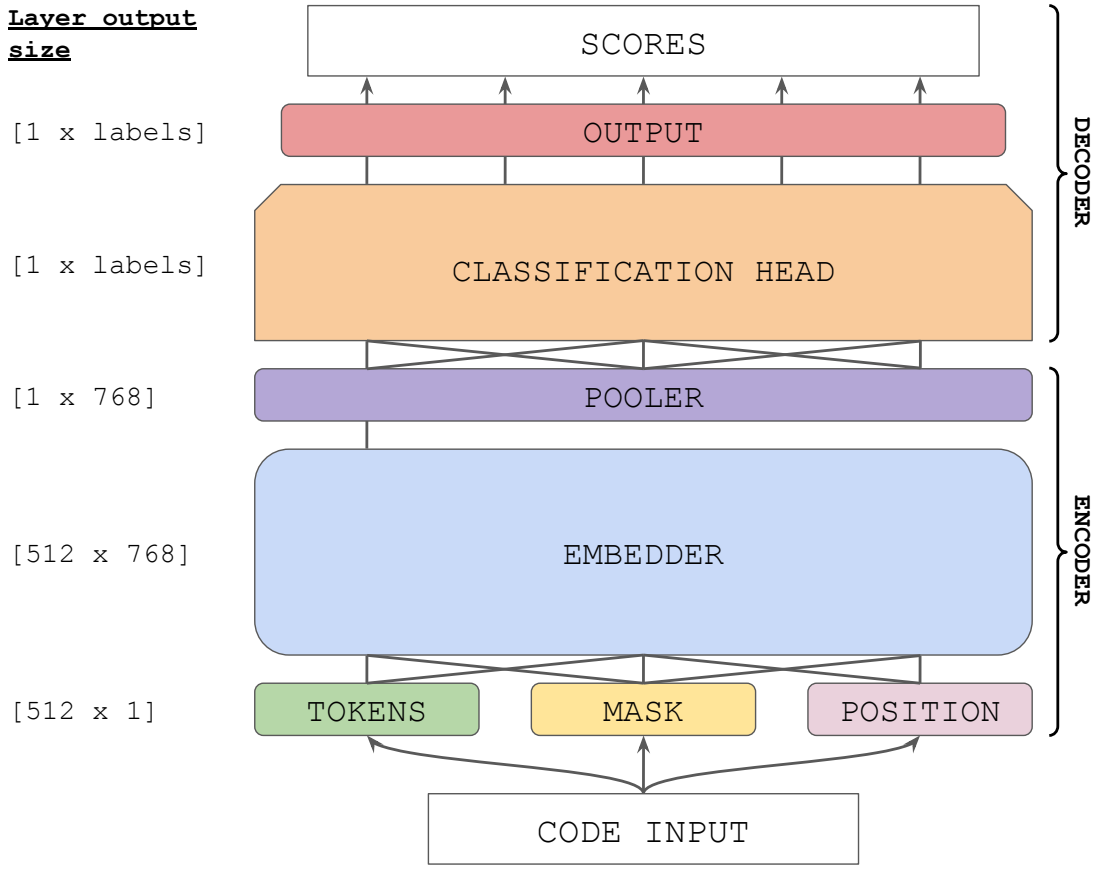


Figure 5.3: Schematic representation of the model and its main components.

"`token ids`", the "`attention mask`" and the "`positional ids`". As explained in the third chapter [3.3], these components are the inputs required by the RoBERTa model.

We decided to truncate each code to the maximal window length of the embedder, i.e. 512 tokens. We chose this strategy over splitting a code snippet into multiple inputs to ensure the dependencies inside the snippet not to be disrupted. For example, if a variable is only defined at the start of the input, the other windows we could construct would miss this assignment, leading to incorrect class predictions. Furthermore, the labelling would be disrupted since some windows would have a label even if there is not the corresponding mutation inside these.

These three tensors are given to the pre-trained GraphCodeBERT model to generate, for each input token, a 768-dimensional vector. For details about the computation inside this `embedder`, please refer to the content of the related Background

sections 3.3 and 3.4. GraphCodeBERT allows one to use textual input in addition to the code tokens. However, we focused here on what the model can learn solely based on the code tokens and the DFG. Regarding the submissions we collected, retrieving the questions' statements and translating them into English would have been complex. Nonetheless, adding these information pieces could improve the model. This will be an aspect mentioned in the *Further work* section [7.4].

However, for our classification procedure, we only need one 768-dimensional embedding to represent the entire student submission. The embedding of the `[CLS]` token is typically selected for this. This embedding extraction is the role of the "`pooler`". The model's default layer applies a tanh activation on the embeddings and picks the output for the `[CLS]` token.

This embedding is then given to the "`classification head`". This block corresponds to a sequence of dense layers converting our 768-dimensional vector into a smaller tensor. The output vector is the prediction `logits` for each class. The `classification head` can contain an arbitrary number of layers, each with its own number of hidden units. The activation function can also be configured with functions like GELU, ReLU or LeakyReLU. Finally, we added the possibility to perform batch-normalization between the dense layers. The impact of previously mentioned parameters will constitute the first experiment inside the next chapter.

The "`output`" layer normalizes the `logits` to assign a score to each possible class. Each value will represent the model confidence in predicting this label. The model will, on a specific input, associate a score between 0 and 1 for each label. Values closer to 1 represent positive label predictions. The model's final output for a student submission is the set of assigned scores, letting the end-user select the most context-adapted threshold. By default, each performance metrics are using a threshold set to 0.5. Values above this are considered as positive predictions.

Referring to the encoder-decoder architecture discussed in subsection 3.3.2, we could describe the same structure here. As depicted in Figure 5.3, the encoder corresponds to creating a single embedding from a source code. The input became a machine-interpretable representation translated back into human-readable scores by the classification head, our decoder here.

The final scores computation can be modified whether we are in a single-label or multi-label configuration. The first requires the scores to sum to one since the model can only predict one of the C possible classes. For this, the last layer computes a $softmax$ on the `logits`. Otherwise, in a multi-label architecture, each prediction score is independent of the others. To obtain this behaviour, the last layer can apply a *sigmoid* function on the `logits`.

The final loss is modified accordingly. In a single-label setting, the model uses a *cross-entropy* loss. For the multi-label configurations, the optimizer relies on a *multi-label soft-margin* loss. The subsection 3.6 covers the formal definitions of these two choices.

5.4 Model implementation

Our model has been implemented as a Python program using the `Pytorch` library [83]. We also used the `HuggingFace` [84] library that stores the pre-trained weights of the embedder on an associated website. `AllenNLP` [86] provided extensive functionalities to work with NLP models on `Pytorch`. It managed the dataset loading, model training and checkpointing, and metrics computation. Finally, `Captum` [82] allowed us to add interpretability computations to the model predictions. We will cover these highlights during the second experiment (**RQ2.5**).

Among the `HuggingFace` models stored on their website[3], we found an interesting pre-trained version of GraphCodeBERT[4]. This version has been released by `Enoch`, alias Ensheng Shi, PhD student at Xi'an Jiaotong University. The provided weights correspond to a feature extraction architecture, i.e. a model generating embeddings from inputs. The official version provided by Microsoft targets fill-the-mask tasks and would have required manual modifications at loading time. Furthermore, `Enoch` has fine-tuned the model on additional Python source codes, which should give this model a more robust representation of the Python language.

Our approach aims to obtain a flexible architecture. To do so, we provided extensive configuration choices through Yaml files. The model also comes with an additional GUI application built with Java. It can connect itself to the Python program and send prediction requests to display them to the user. It can also highlight the interpretability aspects identified by `Captum`. Appendix E shows the GUI appearance and the results it can display.

5.5 Approach & Solution conclusion

This chapter presented the two main components we developed for this thesis: the mutation labelling program and the classification model. The first aims to inject mutations associated with specific misconceptions inside source codes, while the second is our model's architecture. Appendix D presents the 15 classes our model could detect. The *Experiments* chapter [6] will explore these.

[3]https://huggingface.co/models
[4]https://huggingface.co/Enoch/graphcodebert-py

Chapter 6

Experiments

Now our approach has been described, we will delve into the experiments we performed to assess the model performances. The first experiment will explore the impact of the configurable parameters. We will then assess how the model detects misconceptions inside student's submissions.

Every run, unless specified differently, has been executed on the CÉCI infrastructure using the computing cluster named "dragon2". These nodes allowed us to use an NVidia V100 Tesla GPU, speeding up the training process. For more information, we recommend consulting the information page from the organisation[1].

6.1 Model parameters

Finding the best configurable parameters is crucial when building a machine-learning model. These will help the model converge faster, perform better and generalize more on new instances. A broad range of components allows for model tuning: learning rate, batch normalization, fine-tuning, etc. For architecture flexibility, as explained in the previous chapter [5.4], we provided extensive configuration choices. One can modify each model part, from the input generation to the final score predictions. Some parameters, like, for example, classification head structure, can even have an infinite range of possible values.

This configuration freedom comes with a cost: exploring each possible configuration is unfeasible. There are too many possible parameter combinations to allow an exhaustive search procedure. The perspective of finding the best configuration among all the possibilities seems impossible.

Rather than testing each configuration, we could explore how each parameter

[1] `https://www.ceci-hpc.be/clusters.html#dragon2`

impacts the model training and performance. Step by step, by observing the behaviour of each part, we would be able to select individually the best parameters. By combining these, one could expect the final configuration to get close to the best one. However, this conclusion does not always hold in practice. Some parameters rely on others in a specific configuration. Changing one would impact the behaviour of the whole system. Nonetheless, exploring the parameters individually should help us understand their influence on our model.

In this experiment, we decided to study modification to a baseline configuration, the default configuration. We identified it beforehand as a good setting candidate for the mutant classification task. This *baseline* reached good results during preliminary tests with precision and recall scores up to 95% on the test set.

Exploring the parameter influences will consist of finding what modification improves the model behaviour. Conversely, we could find what is unfavourable for our learning process.

When exploring the results beforehand, we identified five research questions that required further analysis. After explaining the parameter modifications we performed, we will analyse these questions in further detail.

6.1.1 Experimental datasets

To train each model, we used the dataset construction we presented in the related *Approach* section [5.2]. From the 13804 student submissions collected, 2500 have been randomly selected. Then, these codes were mutated using the 15 classes we designed (cf. [5.2] and appendix D). After this first mutation process, resulting in 23103 mutated instances, we executed a second time the mutation program. By doing so, we generated 184517 multi-label mutants. We sampled this dataset by selecting, at most, six mutants per original submission, resulting in 14812 instances. To these examples, we added 364 actual failed submissions constituting the `failed` class. This label should cover instances that do not match the misconceptions we designed. Our final dataset for this experiment contains 15176 instances the model should correctly classify.

This dataset has been split into a training, a validation and a test set containing 81%, 9% and 10% of the instances, respectively. These values are classical choices when performing this kind of split. When splitting the datasets, we ensured that all the mutants of an original submission were part of the same set. Having part of the mutants inside the training set and in the test set would have biased the test results.

Appendix F gives more information about the proportion of each class. We can interestingly observe some combinations to be more uncommon. We would expect the model to have less confidence in predicting these labels. The third experiment developed in this thesis will cover this possible limitation [6.2].

6.1.2 Parameter choices and baseline

In the following subsections, *baseline* will reference the default configuration. Each modified configuration will contain one specific change compared to the *baseline*. All the other parts will be the same as the default configuration. This change limit will help us focus on the modified aspects.

However, some parameters are not interesting for the current exploration. We have already broached the embedder in the fourth chapter [4]. The pre-trained version each configuration will use was presented in the previous chapter [5.4]. The embedder also requires other parts, like the tokenizer, to be the same for each configuration. The vocabulary and word indexes should not change since the embedder was trained based on these. The loss metric is not modified since the datasets and objectives are equivalent for each experimental setting.

We identified four interesting parts for parameter tweaking: the input-tensor generation, the embedder, the classification head and the training process.

The first refers to the proportion of code tokens and DFG nodes the model should consider as input. With GraphCodeBERT, we can select how many "`token ids`" are attributed to code tokens or DFG nodes. By modifying the proportion of DFG dedicated entries, this associated sub-experiment will highlight the impact of the graph representation. We could assume some misconceptions to be easily detected, even without code tokens, like, for example, `hardcoded_arg`. However, errors like missing a `return` statement (`miss_return`) will be complex to predict based on the graph only. The DFG does not have an internal representation for this statement, having thus no information about the presence of a `return` or not. The *baseline* will allocate 256 entries for the code tokens, thus 256 for the DFG nodes to sum up to 512 entries. Other choices for the number of code tokens will be 0, 128, 384 and 512.

Secondly, the embedder weights can be either *frozen* or *fine-tuned*. In the first configuration, the classification head is the only model's part that is trained. The weights of the embedder will remain the ones provided by `Enoch` for the task he trained them on. The *fine-tuning* alternative should improve the performance since the embedder would adapt itself to the current prediction task. The power of the $125M$ parameters inside the RoBERTa model should help the classifier get a better

internal representation of the inputs.

The classification head offers multiple complex parameters. One can choose the number of layers, the batch-normalization and the activation function. Since the classification head only contains dense layers, we should not try to stack too many of these. The BERT model should already capture the semantics without needing a more complex transformation afterwards. The more layers we could add, the more the model will have difficulty converging. We will explore three possible layer numbers: 0, 1 and 2. Such values will provide insights into the benefits of making the classification head bigger or relying only on the embedder. When adding layers, we could use batch normalization to see whether this mechanism helps the model generalize better on the test set. Finally, we can explore the activation function choices. LeakyReLU is the default activation function of our model. Such a function avoids completely wiping out high negative activation values. Alternative choices include ReLU and GELU.

Finally, the last explored aspect contains everything related to the training process. We selected two main components: the learning rate and the batch size. The first is crucial for model performance. Assigning it a too-large value is likely to prevent the model from converging. The validation loss would explode, and the model would perform poorly. Setting a too-small value for the learning rate is also unfavourable. The convergence process risks being longer and the model too specific, overfitting the training set. The default learning rate is 10^{-5}, since convergence has been observed to be good with this value. We will consider a larger (10^{-3}) and a smaller (10^{-7}) value to identify if the previous assumptions hold in practice for our model. The batch size choices could also be worth exploring. We opted for 8 instances per batch as the default value. We will try to train the model with batches of 12 and 16 instances.

To account for randomness during the training process, we launched three times each configuration. Each run is associated with a seed to ensure the reproducibility of experiments. These seeds are 42, 33 and 18. We cannot certify these seeds to lead to the best possible model. Other choices could have performed better. However, these should give an overview of the average model performance for each parameter assignment. Using these seeds also ensures the batches contain the same examples in the same order for each configuration.

Each model has been trained for 10 epochs. Setting the same value for each configuration allows us to compare how fast the model converged. It was also observed to be a good compromise between running time and obtained performance.

The following Table 6.1 summarizes the parameters and their selected values to

train our reference model.

Model part	Parameters	Baseline configuration	Other value choices
Input generation			
	- Input window proportion	[256, 256]	[512, 0] ; [384, 128] [128, 384] ; [0, 512]
Embedder			
	- Embedder finetuning	True	False
Classification head			
	- Dense layers number	1	0 ; 2
	- Activation	LeakyReLU	ReLU ; GELU
	- Batch normalization	True	False
Training process			
	- Learning rate	10^{-5}	10^{-3} ; 10^{-7}
	- Batch size	8	12 ; 16

Table 6.1: Reference model table with the parameters studied in this expriment and their default and possible values.

Each performance metric presented in the following subsections has been obtained with a prediction threshold set to 0.5 unless specified differently. Since we are classifying the multi-label artificial dataset containing our previously generated mutants, we need to compute metrics accounting for each class. The precisions, recalls, and f-scores are computed using a micro-average procedure described in the related *Background* section [3.7]. The multi-label ROC curves use the same mechanism. In the following graphs representing these curves, float numbers inside the legends will correspond to the AUC scores. The diamond symbol on loss curves represents the best epoch, while on ROC curves, it depicts the model's performance with the default prediction threshold (set to 0.5).

The loss metric will be the multi-label soft-margin loss presented in the *Background* chapter [3.6].

6.1.3 Results

Before delving into the modifications, we will present the performance obtained with the *baseline* configuration.

First, we need to check if the learning procedure reached its objective, i.e. lowering the loss as much as possible. If our training lacks this observation, we could question

the parameter choices and assume a misconfiguration. The following Figure 6.1 presents the learning curves for each run.

The model behaviour does not vary significantly from run to run. The training process reached minimal loss in the neighbourhood of the last epoch. Table 6.2 highlights these comparable behaviours. The *baseline* configuration performs well on the test set of the mutant classification task.

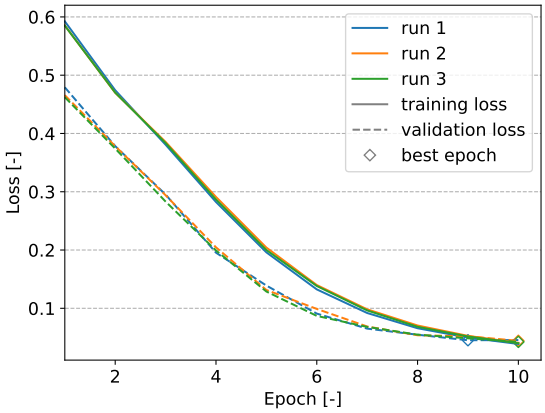


Figure 6.1: Multi-label soft margin loss curves for the baseline configurations.

Metric	Run 1	Run 2	Run 3
Precision			
	0.95324	0.95308	0.94190
Recall			
	0.92547	0.92209	0.92818
Fscore			
	0.93915	0.93733	0.93499

Table 6.2: Scores obtained by the *baseline* on the test set.

Appendix G summarizes the f1-score metrics of each configuration modification. One can observe the *baseline* to be part of the best configurations. We can also see that some modifications, like changing the activation function, did not disturb the model as we would expect. Only some configurations created performance gaps above 50%. Table 6.3 shows the configuration subset which impacted the model the most.

Model part	Configuration	Run 1	Run 2	Run 3
	baseline	0.93915	0.93733	0.93499
Input generation	`[0, 512]`	0.32725	0.34241	0.34548
Embedder	No fine-tuning	0.60205	0.60382	0.60050
Training process	lr 10^{-3} + batch 8	0.20114	0.21573	0.21573
	lr 10^{-7} + batch 8	0.30369	0.30316	0.29851

Table 6.3: Subset of Table 8.3. Presented configurations are the one we identified as the farthest from the *baseline*.

Considering these results, we identified five research questions (RQ) that the rest of this section will cover:

- **RQ1.1**: Are all the labels equally complex to predict for the *baseline*?
- **RQ1.2**: What did the DFG miss to obtain these results?
- **RQ1.3**: Are the model predictions driven by the embedder?
- **RQ1.4**: Is fine-tuning mandatory to get better performance?
- **RQ1.5**: How was the learning process impacted by the learning rate?

When answering these questions, we will, unless specified differently, present the results of the first seed. We performed this choice since the results of the two other seeds lead to equivalent discussions and conclusions. Appendix H gathers the graphs we did not display in the rest of this section.

RQ1.1: label complexity

This question relates to the scores above 0.9 the *baseline* obtained. We asked ourselves whether the model performed equally to detect the different injected errors. All misconceptions are not equally present, and some should be harder to identify. Figure 6.2 represents label-wise ROC curves obtained by the first run.

Two classes stand out: `miss_return` and `hardcoded_arg`. The fact that our model performed well on these labels is not surprising. `hardcoded_arg` relies on a specific mutation where one function argument is reassigned as the first instruction. By looking at the two first lines of a submission, the model should already have enough information to handle this class. The simplicity of this mutation could constitute a limit that will be discussed during the second experiment (**RQ2.2**).

`miss_return` appears when all `return`s are replaced with a `print` or one is removed from the source code. The first case is more straightforward since the model could check the absence of return statements (see Listing 6.1). For the second, the model should understand why a specific position requires a return (see Listing 6.2). However, most of the time, since the functions are limited in size, we could infer this class by looking at the last instruction. Students mostly end their function bodies with a return statement.

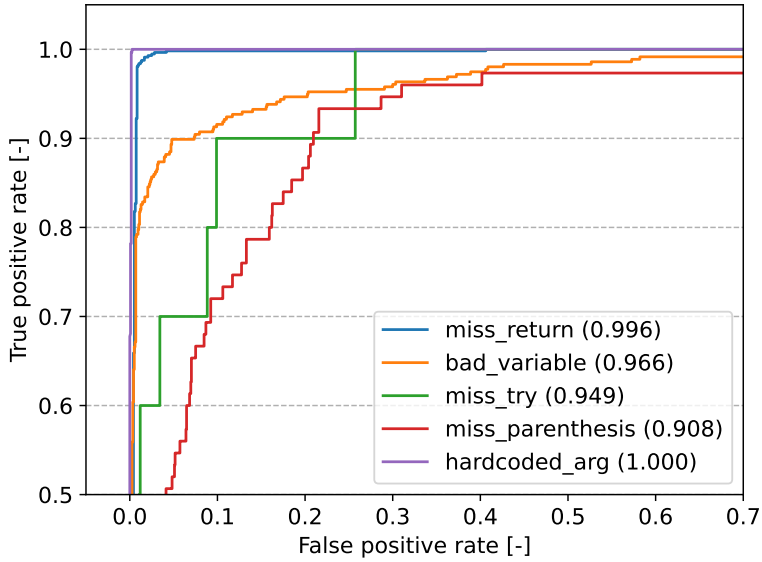


Figure 6.2: Label-wise ROC curves from the first run of the *baseline* configuration. A subset of the 15 designed labels has been selected to highlight specific observations. The other labels obtained AUC scores larger than the value obtained for `miss_parenthesis`. The axes have been truncated to get a closer look at the differences around the upper left ROC-space corner.

```
def get_max(lst, step):
    if len(lst) == 0:
        print(None)
    lst = lst[::step]
    out = max(lst)
    print(out)
```

Listing 6.1: The model could classify these cases better since checking for return presence is sufficient.

```
def get_max(lst, step):
    if len(lst) == 0:
        return None
    lst = lst[::step]
    out = max(lst)
```

Listing 6.2: More complex situation, the model should understand where the return is required.

Our model misclassifies more `bad_variable` instances. This class contains two types of errors: either the student used an unreferenced variable or they replaced an existing variable with another. The first situation should be more easily detectable. Indeed, the DFG representation, along with the code tokens, should contain information about a variable that is used but was never assigned to a value beforehand. However, the second case is trickier. The model needs to know the required variable to infer it to be misplaced. For example, one could not decide if `a = a*b` contains an incorrect variable. But if we know the next instruction to be

`return b`, we could infer the first occurrence of `a` to be incorrect.

The fact that `miss_try` obtained a lower score connects with this class's representativeness. Students should put their file-handling code sections inside try-except blocks to obtain perfect submissions. But not all of them do so and, Inginious does not always check such cases. The `miss_try` label only appears 102 times among the 12294 training instances, in less than 1% of the generated set. This lack of representativeness could explain why the model is less performing for this label. Since this class assumes replacing a try-except block with its try body, there is no difference between mutated instances and successful submissions without try-except. The second experiment [6.2] will look at how the model predicts these misconception cases. These performance scores could also relate to the small number of test instances (10 out of 1526) with such a label. As depicted on the label-wise ROC curves, the model can detect eight instances with an FPR of 10%, corresponding to more than 100 examples. If we check the label-wise scores with our default prediction threshold, we obtain three hits and seven misses, leading to 1 for precision and 0.3 for recall.

Finally, the `miss_parenthesis` label seems to be the most complex class to predict. Indeed, this phenomenon also relates to the mutation limitations our dataset contains. This class refers to the removal of associative parentheses inside students' submissions. However, some students appear to overuse these. For example, they could write `(a*b)*c`, maybe for simplifying their reasoning or readability reasons. Nonetheless, other students would write `a*b*c` for the same question. After mutation, the first case becomes `a*b*c`, like the second, but associated with a different label. The model receives contradictory information and should be less confident in predicting this class. This conclusion corresponds to the ROC curve behaviour. The label-wise scores fall to 0.67 for precision and 0.32 for recall, making this class the lowest-performing one. Real error cases, like using `a+b/c` instead of `(a+b)/c`, should also be misclassified. The second experiment [6.2] will look at how our model handles these classes on actual students' mistakes.

Conclusion: The mutations we injected are not equally understood by the model. The easiest mutations, `hardcoded_arg` and `miss_return`, are also the easiest classes for the model. Less represented labels are also more difficult for the model. Finally, cases that could create contradictory examples are less correctly handled by the model.

RQ1.2: DFG impact

From the four additional splits we tried [2], only the `[0, 512]` configuration obtained a different range of scores on the test set. The objective of this question is to identify what caused this performance gap. We already discussed the cases that could be missed based solely on the DFG. The purpose of this section is to explore if our assumptions are confirmed. Figure 6.3 displays the label-wise ROC curves for the first run of the `[0, 512]` configuration.

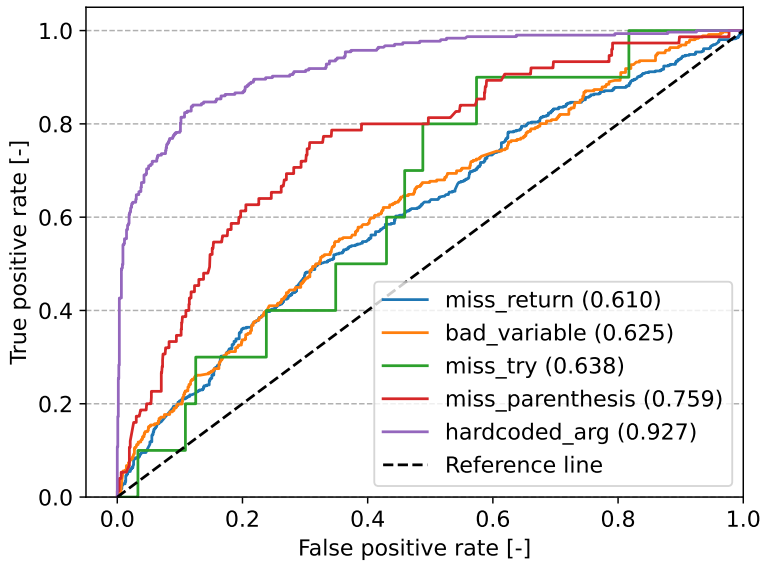


Figure 6.3: Label-wise ROC curves from the first run of the `[0, 512]` configuration. The selected labels are the same as for Figure 6.2.

As previously explained in the subsection 6.1.2, completely removing the code tokens misses misconceptions. For example, return statements are absent from the DFG representation. Predicting if the student uses all the required returns becomes unfeasible.

As we could expect, we can observe that the `hardcoded_arg` label obtains the best performance. Our model, even without the code tokens, managed to understand this injected misconception. However, it seems to be less confident in its predictions; the associated curve is right-shifted, indicating more false positives. We could conclude that the model has less learning room to understand this misconception

[2] To obtain these configurations, we modified the behaviour of the original input generation from Microsoft. Their code initially ignored nulled parts and filled all the available entries inside the 512 slots window.

class. The code tokens are more important to help the model predict the mutants' labels.

However, one could be surprised by the model behaviour for the `bad_variable` label. We would have expected the model to better understand the misuse of a variable. Our model, with the complete data flow, should have everything required to predict what constitutes this error class. However, the DFG construction we presented in the *Background* section [3.1.2] does not contain enough information for these cases. If a variable is never assigned or used, then this variable will be missing from the DFG. For example, in the following code snippet (Listing 6.3) where $\texttt{b}^7$ has been replaced by $\texttt{a}^7$:

```
def sum(a⁰, b¹):
    tmp² = 0³
    out⁴ = tmp⁵ + a⁶ + a⁷
    return out⁸
```

Listing 6.3: `bad_variable` miss.

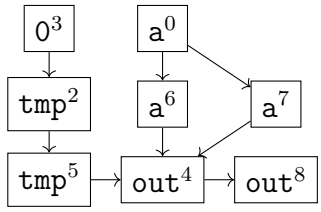


Figure 6.4: Constructed DFG.

One can observe that the $\texttt{b}^1$ variable is not represented in the DFG (Figure 6.4). Relying solely on the DFG, we could conclude that each assigned variable has been used to compute $\texttt{out}^4$. Missing the information about the $\texttt{b}^1$ variable misleads the model.

Furthermore, the DFG is not given as this to the model. As explained in the corresponding *Background* section [3.4], the DFG is encoded inside the `attention mask`. The input construction assumes the presence of the code tokens to represent the variable's names and positions. If we take back the example of Listing 6.3, the model receives the following tensors as input (truncated to the effective window):

```
token ids:
    [CLS, SEP, PAD, PAD, PAD, PAD, PAD, PAD, PAD, PAD]

attention mask:
   [[  1,   1,   1,   1,   1,   1,   1,   1,   1,   1],
    [  1,   1,   1,   1,   1,   1,   1,   1,   1,   1],
    [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0],
    [  0,   0,   0,   0,   1,   0,   0,   0,   0,   0],
    [  0,   0,   0,   0,   0,   0,   0,   0,   0,   0],
    [  0,   0,   0,   0,   0,   0,   1,   1,   1,   0],
    [  0,   0,   0,   1,   0,   0,   0,   0,   0,   0],
```

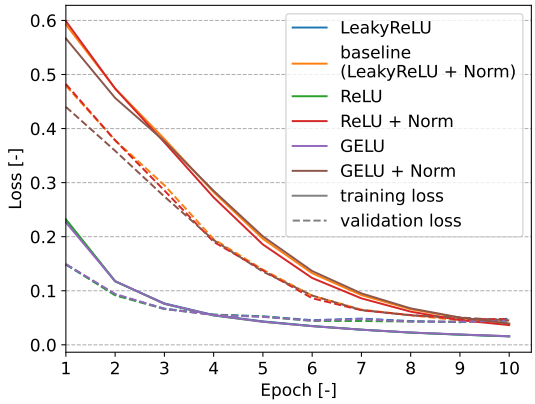
```
    [  0,   0,   1,   0,   0,   0,   0,   0,   0,   0],
    [  0,   0,   1,   0,   0,   0,   0,   0,   0,   0],
    [  0,   0,   0,   0,   0,   1,   0,   0,   0,   0]]

positional ids:
    [  2,   3,   0,   0,   0,   0,   0,   0,   0,   0]
```

These tensors do not contain enough information. The model lost the `token ids` of the nodes inside the DFG. While we could infer some information from the `attention mask`, we need additional knowledge to be confident in our predictions. This observation explains why the model behaved poorly, even on the easier labels.

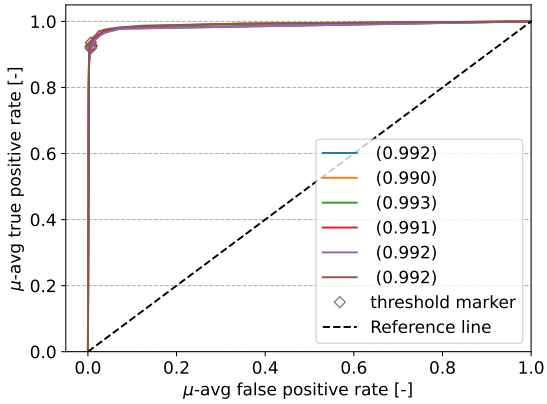
Conclusion: There is an undeniable gain to using the code tokens; they are the most crucial part of our model inputs. The DFG construction needs them to represent the variables and add information about the operations performed. However, when varying the other configuration parameters, the DFG could help the model catch some cases. Keeping a balanced proportion of code tokens and DFG nodes seems an acceptable compromise to benefit from the aspects they respectively represent.

RQ1.3: classification head importance

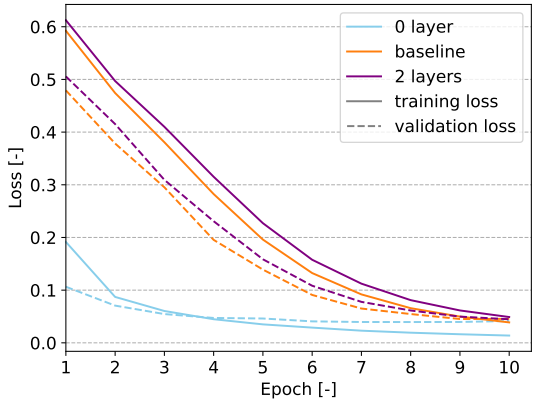
Modifying the head while fine-tuning the embedder, as initially designed in this experiment, does not lead to significant differences between models. Keeping the same number of dense layers while modifying the activation function and batch normalization gives equivalent results to the *baseline* (see Figure 6.5b). The only observed change relates to the higher convergence speed for configuration without batch normalization. However, as depicted in Figure 6.5a, each model reached nearly the same validation loss at the end of the training process. These parameters, in this experimental context, are less crucial.



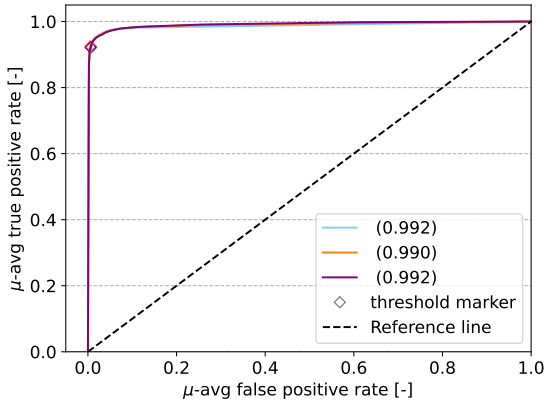
(a) Loss for norm+activation changes



(b) ROC curves for norm+activation changes (use the same color legend as Figure 6.5a)



(c) Loss for layer changes



(d) ROC curves for layer changes (use the same color legend as Figure 6.5c)

Figure 6.5: Results for the first run of the *classification head* sub-experiment.

Nonetheless, we would have expected to observe more variations when modifying the number of dense layers. When considering the results in Figures 6.5c and 6.5d, we asked ourselves if the "decoder" complexity is important in our architecture.

Since changing the number of parameters inside the classification head leads to the same results, we assumed the $125M$ embedder weights to drive the model learning and predictions, having less room for the upper model layers. Freezing the embedder should highlight how the classification head complexity impacts the model performance. Figure 6.6 shows the model performance when the embedder weights are frozen. We tried these configurations with the same seeds and datasets as the other sub-experiments. The f1-scores for each run lie in Table 6.4.

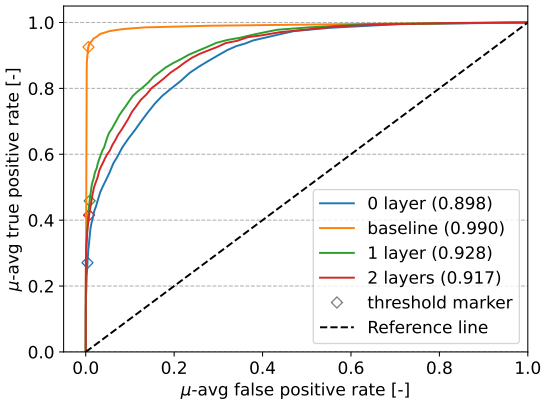
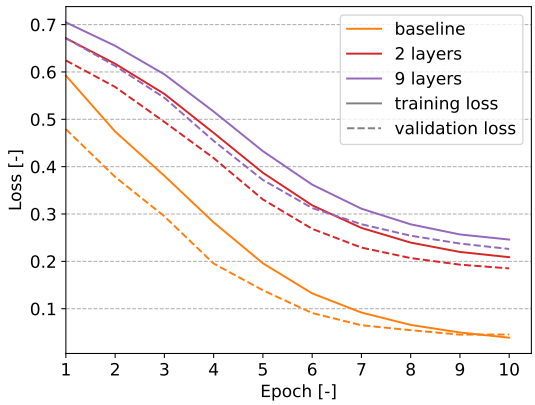


Figure 6.6: ROC curves changing the number of layers without fine-tuning the embedder.

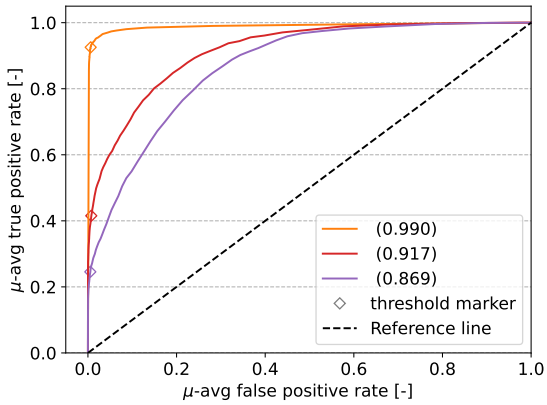
Model	Run 1	Run 2	Run 3
0 layer			
	0.41647	0.41502	0.41799
1 layer			
	0.54499	0.56037	0.56724
2 layers			
	0.56498	0.56377	0.56746

Table 6.4: F1-scores obtained without fine-tuning the embedder.

We can observe each additional layer to improve the model performance. However, this gain starts to saturate with one layer; the improvement pace slows down. Pushing the limits further to see if adding complexity to the classification head does not improve the model, we explored a configuration with 9 additional dense layers. We already discussed that adding more layers should make convergence harder and reduce the performance. Figures 6.7a and 6.7b compare the *baseline* and configurations with 2 and 9 additional layers.



(a) Loss curves



(b) ROC curves (use the same color legend as Figure6.7a)

Figure 6.7: Comparison of the first runs without fine-tuning the embedder and using up to 9 dense layers.

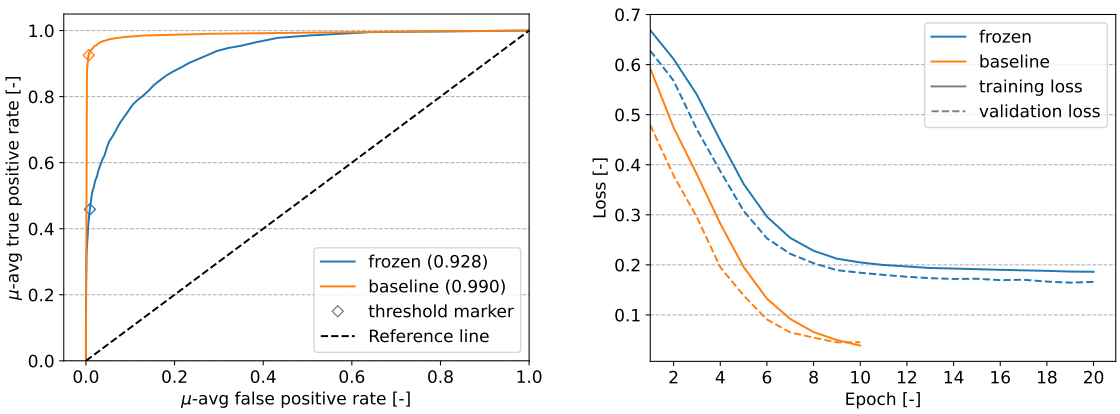
Considering the lower AUC score for the configuration with 9 layers compared to

the one using 2 layers, we can conclude that adding more dense layers makes the model converge poorly. The classification head is not as powerful as the embedder, which uses Transformer-encoder blocks.

Conclusion: The classification head is not the model part which internally understands the injected mutations. This model sub-part can stay simple, i.e. use up to two additional dense layers. The power of our model on this classification task comes from the embedder and its $125M$ parameters. However, training only some dense layers would have spared some resources and time. Experiments in section 6.2 will highlight if the bad behaviour is also observed on real submissions. The next sub-experiment will explore whether fine-tuning is mandatory or whether the training process is not adapted.

RQ1.4: fine-tuning gain

Ignoring the embedder fine-tuning would speed up the training process. Going from 75 minutes to 28 to train a model is a welcome gain in time and energy resources. The *Further work* section [7.4] of the *Conclusion* will present a preliminary analysis of the environmental impact our training process could have. Due to the time limit, detailed discussion should be part of further experiments. However, as discussed for **RQ1.3**, the classification head alone is not powerful enough. Figure 6.8a presents the ROC curves comparing fine-tuning and embedder freezing configurations.



(a) ROC curves

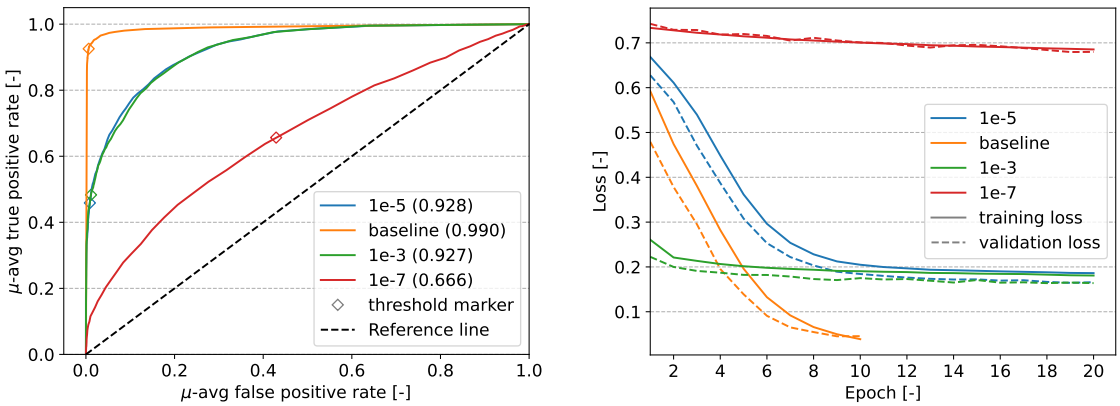
(b) Loss curves

Figure 6.8: Results for the first run of the *embedder fine-tuning* sub-experiment.

We can observe a gap between the *baseline* and the frozen configuration. Apart from the classification head limits we covered in the previous section, we could obtain better results by modifying the learning process. We first hypothesized the convergence to reach a temporary plateau. The 10 epochs we fixed at the start of

the experiment could be a limit. Figure 6.8b shows the learning curves we obtained when adding 10 additional epochs.

As one can observe, there is a convergence speed change near the 10th epoch. The loss continues to decrease at a very slow rate. We could add more epochs to see if this decrease reaches the same level as with fine-tuning, but it would be contrary to our training time reduction objective. We supposed our learning rate to be not adapted when training only this model sub-part. Figure 6.9a shows the ROC curves we obtained when testing two other learning rates: 10^{-3} and 10^{-7}. We performed these training processes three times with the same seeds and datasets as the other sub-experiments. The results obtained for the other runs lie in Appendix H.



(a) ROC curves

(b) Loss curves

Figure 6.9: Results for the first run for the learning rate modifications when freezing the embedder weights.

As depicted in Figures 6.9a and 6.9b, the learning rate choice does not allow the model to perform as well as the baseline. Choosing a too-small learning rate (10^{-7}) leads to high loss values (above 0.65). Reducing it to reach the same level as the baseline would take a very long time. Using a too-high learning rate (10^{-3}) does not drastically change the learning process. The loss reached the lower values faster, but the same behaviour as our original learning rate is observed after the 10th epoch. When the classification head is trained alone, 10^{-3} does not disrupt much the performances. The two curves reach the same levels. The following research question, **RQ1.5**, will explore the impact of this rate when the fine-tuning procedure is activated.

However, we see that the learning rate is not the key to performance for the classification head alone. This model part is not powerful enough in itself. Adding more epochs, once again, could help the model reach lower loss values. However,

since our objective was to diminish the learning heaviness of the model, doubling the number of epochs would not fulfil this goal. Alternatively, we could explore if reducing the number of epochs when fine-tuning leads to results comparable to the *baseline*.

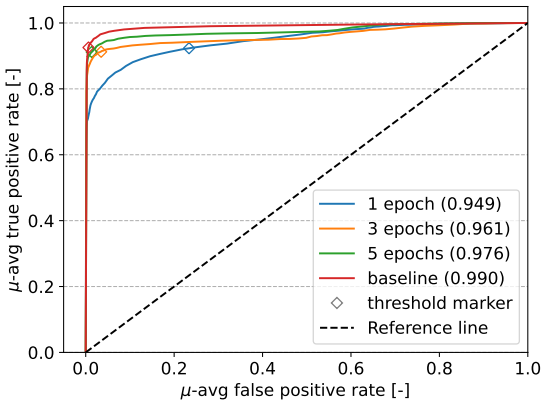


Figure 6.10: ROC curves changing the number of epochs with embedder fine-tuning.

Model	Run 1	Run 2	Run 3
1 epoch			
	0.50982	0.55503	0.52549
3 epochs			
	0.84266	0.84652	0.87484
5 epochs			
	0.90731	0.92040	0.92241

Table 6.5: F1-scores obtained with embedder fine-tuning.

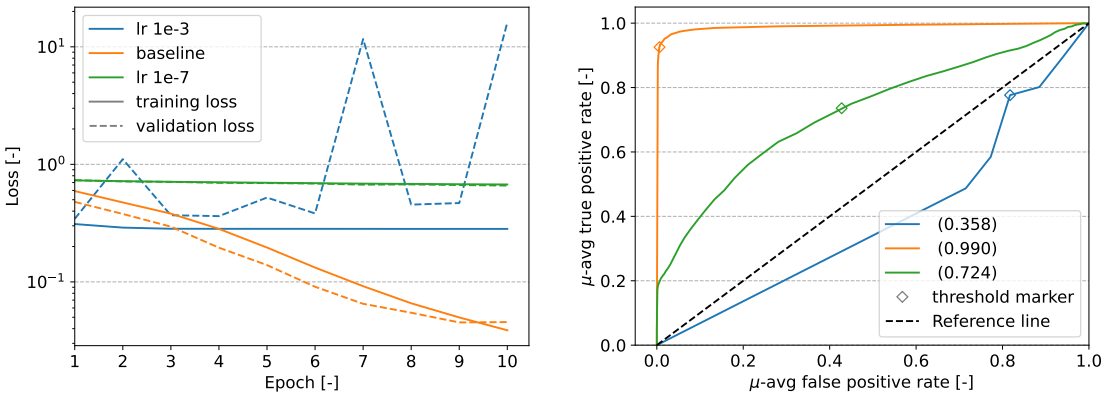
In Figure 6.10 and Table 6.5, one can observe the results to be better than without fine-tuning the embedder, even if the learning has not yet converged to the lowest possible loss. Limiting to 5 epochs provides f1-scores close (less than 4%) to the best scores obtained with the 10 epochs model and divides the training time by two. The process ends after 30 minutes, as we managed to do when freezing the embedder for 10 epochs.

Conclusion: Embedder fine-tuning is crucial to obtain the best performance on our artificial task. The classification head lacks the learning power to represent the submissions, and its predictions are not as robust as we may want. However, even with limited training, fine-tuning the embedder outperforms what we could obtain by freezing it. Simplifying the training heaviness can be performed by limiting the epochs number rather than the fine-tuned parts. The second experiment [6.2] will check if the performance of the limited training model aligns with the *baseline* on students' submissions.

RQ1.5: learning rate impact

When discussing **RQ1.4** in the previous section, we observed the 10^{-3} learning rate not to disrupt the model without embedder fine-tuning. We wanted to know the behaviour of GraphCodeBERT when modifying the learning rate to equivalent

values. The performance we highlighted in Table 6.3 shows the impact of this choice on the model behaviour. Figure 6.11a shows the loss curves obtained when changing the learning rate. Figure 6.11b displays the corresponding ROC curves.



(a) Loss curves (y-axis log scaled)

(b) ROC curves

Figure 6.11: Results for the first run of the learning rate change sub-experiment.

At this thesis's beginning, the model obtained bad results even with fine-tuning. After investigation, we found the optimizer to use a learning rate equal to 10^{-3} as the default value. Once we identified this, we used 10^{-5} instead and obtained the *baseline* results we presented previously. Analyzing the results we obtained in this research question highlights what happened. Using 10^{-3} when fine-tuning the embedder causes the validation loss to explode (goes above 10) during training. Its lack of correlation with the training loss proves the model does not learn how to classify the instances.

Conclusion: When using GraphCodeBERT, the learning rate should be carefully considered. This observation could also explain why CuBERT did not perform correctly during our preliminary study [4]. 10^{-5} has been found to lead to good results for our model. Hence, we selected it as our default value.

6.1.4 Conclusion

This experiment allowed us to test our training process. The following section will check if this training task is valid to obtain good predictions on students' submissions.

The experiment developed in this section also provided an analysis of the impact of model parameters. The DFG, by its limited representation power, is not powerful enough without the code tokens. The graph requires these to represent part of

its information inside the attention mask. These representations could be part of improvement tracks, for example, to include unused constants or arguments. The classification head is also not powerful when trained alone. Either when modifying its complexity or changing the learning process, we highlighted that the embedder is the part making the model obtain f1-scores above 0.9. The classification head can remain simple, i.e. limited to at most two layers. However, when fine-tuning, the learning rate choice is crucial. Otherwise, the validation loss could explode, meaning the model would not understand the examples, or it could never converge, meaning the model would require intensive training to catch up with the performance obtained by the *baseline*.

This experiment also allowed us to highlight specific mutations that are more difficult to detect for the models. These cases include less-represented classes or ones creating contradictory examples. `miss_try` and `miss_parenthesis` are labels we identified with these behaviours. The following experiment will explore the classes corresponding to more complex cases for actual students' failed submissions.

6.2 Model validation and dataset size

At this stage, we have a model with a *baseline* configuration performing well in classifying the mutants we generate. The aspect we need to explore now is: "Could the model, with its *baseline* configuration, help students?"

To answer this question, we will use the model weights learned from the first experiments. This choice means the architecture used the training dataset described in section 6.1.1. However, we will adapt the test set. The objective is no more to see whether the model understands the mutations but whether it generalizes well on students' mistakes.

After explaining our methodology for creating the test instances, we will try to answer five new research questions:

- **RQ2.1**: Are the *baseline* scores for the mutant task a reflection of the performance for students' submissions? Are less-performing configurations also bad for feedback to students?
- **RQ2.2**: What does the model manage to understand? What are the most complex cases?
- **RQ2.3**: Are larger datasets helping the model to predict better?
- **RQ2.4**: Is intensive training required? Is using fewer epochs acceptable?

- **RQ2.5**: Are interpretability techniques useful?

6.2.1 Manual test-set labelling

To construct the set on which the fine-tuned models will be evaluated, we need to find students' failed submissions. Instead of using the 23 Inginious questions we selected for mutation, we will use exam submissions. We chose this strategy for three reasons.

First, students are treated equally and objectively. They had the same preparation time, were in the same room and had access to the same resources during the exam. Selecting exam submissions should reflect personal students' misunderstanding after their study process.

Secondly, the exam submissions do not overlap with our training set. Questions could look like semester exercises on Inginious but are not the same. The model should not have already seen the examples during its training process. This strategy will allow us to explore model generalizability.

Finally, tests have been extensively designed to evaluate exam submissions. The unit tests deployed for the exams should be robust to ensure fair and unquestionable grades. We assume that no failed exam submissions have been considered successful.

We selected 500 submissions at random among the 121428 failed submissions collected from previous years. By carefully analysing each instance and removing empty, uninteresting and duplicate ones, we constructed a dataset of 161 examples. Each has been manually labelled using the 15 misconception classes we designed for the mutation (cf. Appendix D). We also used the "`failed`" label to represent undecidable cases. By doing so, we highlighted 302 mistakes inside the submissions. Appendix I shows the statistics behind this evaluation dataset.

Some students' submissions were surprising and nearly unclassifiable: some because students' solutions were complex and highlighting a specific problem was hard, others because there were too many problems to understand what the student misunderstood. This observation is already somehow a limit to our model. The students' misconceptions should fall into the cases we designed, without the flexibility a human rater could demonstrate.

6.2.2 Selected configurations & test process

Regarding the results of the *baseline* of the first experiment on the mutant classification task, we decided to use it as this experiment *baseline*. Since we saved

the *baseline* weights from the first experiment, we can load these and launch the evaluation process for them. Reusing these weights would spare computing and time resources.

We measured each performance metric three times, one for each of the seeds from the previous experiment. The precisions, recalls and, f1-scores also use the micro-averaging technique we presented in the *Background* chapter [3]. The prediction threshold is, by default, equal to 0.5, as previously mentioned.

RQ2.1 and **RQ2.2** will use the *baseline* models and explore their results on the failed submissions. **RQ2.1** will also explore the models from the first experiment where embedder fine-tuning was disabled. **RQ2.3** will cover new training datasets and how the *baseline* configuration behaves with these. **RQ2.4** will use the epoch-limited models from **RQ1.4**. Finally, **RQ2.5** will explore briefly the interpretability outputs.

6.2.3 Results & discussion

As for the first experiment, we will focus on the results of the first seed unless specific differently. Appendix N will gather the graphs we decided not to display here but which lead to the same observations and conclusions.

RQ2.1: *baseline* and training quality

Figure 6.12 and Table 6.6 show the *baseline* performance on the evaluation dataset. Since students' submissions are more complex than the injected mutations, we expected this configuration to obtain lower scores than those achieved on the training task. **RQ2.2** will cover the classification limits the model demonstrates. However, we first need to check the usefulness of the training task. If less-performing models from the previous experiment perform better than the *baseline* here, we could conclude that training on injected mutations does not enable the detection of students' mistakes. Figures 6.13a and 6.13b compare the *baseline* and the frozen-embedder variants.

As we can see, the *baseline* obtains an average recall and f1-score nearly two times higher than the other models. An equivalent performance scores gap is depicted in Figures 6.14a and 6.14b, exploring learning rate changes.

While using a learning rate equal to 10^{-7} leads to recall values within 4% of those obtained by the *baseline*, this comes with a cost on the precision score following the classical precision-recall tradeoff. Comparing the f1-scores, as they are accounting for this tradeoff, highlights this configuration is the less-performing one. We

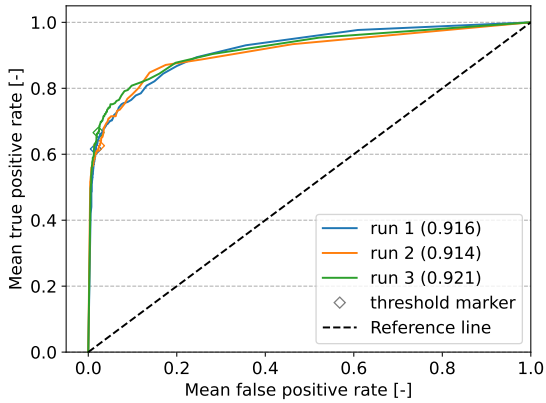
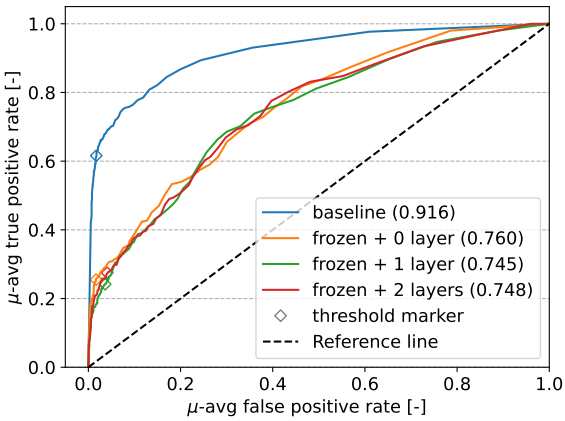


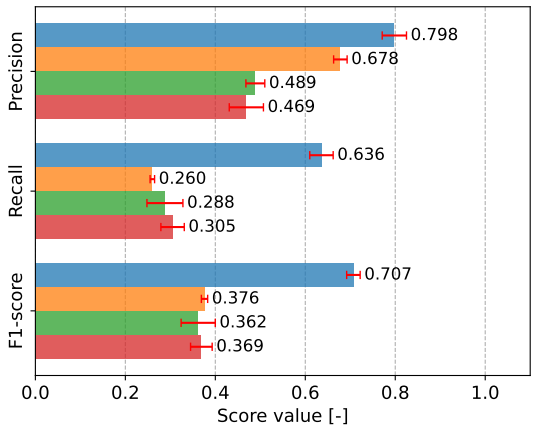
Figure 6.12: ROC curves for the three *baseline* models.

Model	Run 1	Run 2	Run 3
Precision			
	0.82667	0.77459	0.79134
Recall			
	0.61589	0.62583	0.66556
F1-score			
	0.70588	0.69231	0.72302

Table 6.6: Performance metrics for the three *baseline* models.



(a) ROC curves for the first seed

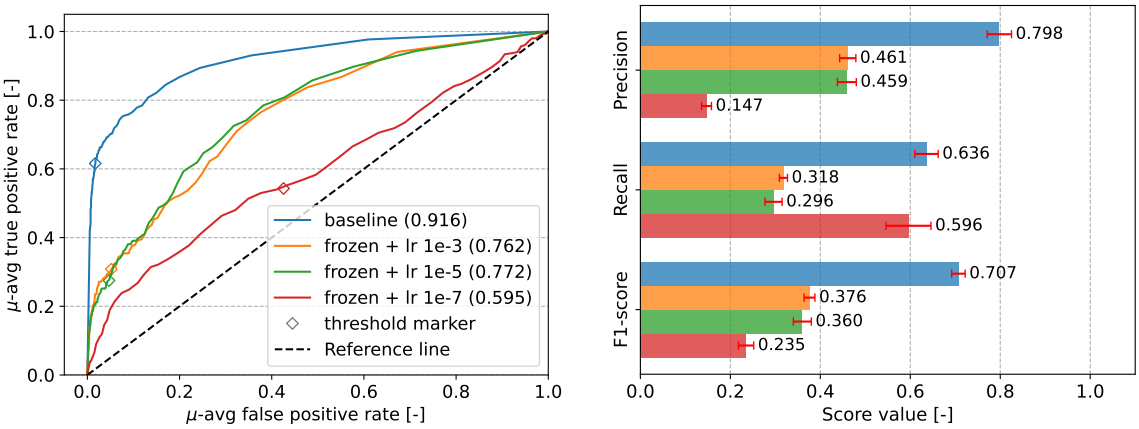


(b) Global bar plot (use the same color legend as Figure 6.13a)

Figure 6.13: Results for the models without embedder fine-tuning when modifying the number of classification head layers.

can conclude, based on these results, that the training task is valid. The model performance on the evaluation dataset links to the loss each model obtained at the end of its training process. The *baseline*, which reached the lowest loss, is the best-performing one when predicting actual misconceptions.

Conclusion: Performing well on the training task comes with better performance on the evaluation dataset. This observation asserts that the training objective, i.e. finding mutants inside an artificial dataset, represents the final goal. Parameters that are good for this task seem performing for actual failed examples. Maybe the lower performance observed for the *baseline* relates to the mutations rather than the



(a) ROC curves for the first seed

(b) Global bar plot (use the same color legend as Figure 6.14a)

Figure 6.14: Results for the models without embedder fine-tuning when varying the learning rate.

training task: the model could understand cases close to designed mutations, and slight changes could perturb the predictions. The following section will highlight what the model lacks and what could improve its behaviour.

RQ2.2: false alarms and misses

This question will explore the label-wise performance of the *baseline* models on the evaluation dataset. Appendix K shows the average scores the models reached for each class. By considering the results in Table 8.6, we highlighted three categories.

First, there are the classes for which the models get high scores, i.e. average f1-scores above 0.8. These values correspond to `miss_return`, `print_return`, `bad_assign` and `bad_loop`. We could highlight that these labels are associated with more precise mutations that are more likely to produce erroneous codes. These classes represent actual mistakes students make and do not require to be extensively modified to include more cases.

Secondly, we have the labels obtaining average f1-scores between 0.4 and 0.8. The labels belonging to this group are more difficult to determine manually. For example, `bad_range` includes calls to the range function with an additional `+1` (see Appendix D). However, as depicted by Listing 6.4, this addition could be required by the question's statement. The model does not handle these cases properly; it over-fitted the `+1` presence to predict this class. This observation is also valid for the `hardcoded_arg` label. Listing 6.5 shows an instance where we manually detected

an occurrence of the label. However, none of the three *baseline* runs identified it. We believe this behaviour to be characteristic of mutation over-fitting. The model has over-learned the injected instructions rather than the misconceptions' meaning. `hardcoded_arg` reassigns an argument to an integer as the first instruction of the function body. Since this case does not use an integer, the model does not predict this class. It expects that the instances are closer to the mutants it learned during the training task.

```
def produits(n):
    out = []
    for r in range(1,n+1):
        if n % r == 0:
            out.append((r, n//r))
    return out
```

Listing 6.4: Correct example for which the `+1` inside the call to `range` is required.

```
def combien(n):
    n=input(int())
    if n>0:
        n=n
        print(n)
    if n>0 and i<=n:
        k=range(0,n+1)
        i+k=n
        print(i+k=n)
```

Listing 6.5: Instance number 141 of the evaluation dataset. No model predicted `hardcoded_arg` for this example.

This mutation over-fitting is a limitation of our model. Even if the model obtained f1-scores above 0.9 for the mutants, it does not generalize on cases that are slight changes to these. This observation is absent from the first highlighted category since these misconceptions are more specific. When a student uses a print instead of a return, there are fewer alternative ways to write this mistake. `hardcoded_arg` suffers from the representation limits of the mutation. Placing the instruction somewhere else or not using an integer will change the model predictions since these specific cases conditioned it.

Finally, the last group contains the labels for which the model obtained f1-scores lower than 0.4. We highlighted `miss_try` as a more difficult label during the first experiment. The limited number of training instances, associated with the confusion created by successful submissions not using try-except blocks when they should, are possible explanations for the lower performance of the models for this label. Listing 6.6 shows an example of a misclassified instance for this class. `miss_parenthesis` is also a case we identified as a complex mutation when discussing **RQ1.1**. However, the limited number of instances in the evaluation dataset (2) does not allow us to conclude how the model would have performed globally. Nonetheless, since this label suffers the same limits as `miss_try`, i.e. mutations

that are equivalent to unmodified instances, we could expect the model not to predict these optimally. Listing 6.7 represents one of the two instances the models missed.

```
def load_students(filename) :
    d ={}
    with open(filename) as file :
        txt = file.readlines()
        txt = line.strip("")
        for line in txt :
            line = line.split(",")
            print(line)
```

Listing 6.6: Instance number 27 of the evaluation dataset. No model predicted `miss_try` for this example.

```
def serie(n) :
    count = 0
    if n == 0:
        return 0
    else:
        for i in range(1, n+1):
            count += -1**(i-1)//i
    return count
```

Listing 6.7: Instance number 5 of the evaluation dataset. No model predicted `miss_parenthesis` for this example.

The fact that `miss_try` obtained scores this low, compared to the other labels, could also be linked with the scores that `bad_file` obtained. We can observe in Table 8.6 that these lie in the lowest-performing group. Our current hypothesis to explain this phenomenon comes from the selected questions and the representativeness of these classes. Table 8.2 shows that `bad_file` only appears in less than 2.4% of the training instances. This proportion gets even smaller with `bad_file` appearing in 0.83% of the same set. Since we selected only four exercises about files to generate our mutants (see the questions of Session 6 in Appendix C), The model did not have enough information to predict these correctly. Maybe our mutations were correct and representative, but the model had no chance to learn them robustly when overflowed by the other misconceptions.

We highlighted two other limits from the model behaviour: exercises' contexts and labels' saturation. The first case refers to the contexts of the exercises we selected for the training task (see Appendix C). When selecting these, we ignored exercises focusing on Python objects and classes, as discussed in section 5.2. However, when using the model on exam questions on these concepts, like in Listing 6.8, the model predicts `miss_return`. Since it did not see during fine-tuning that `__init__` methods are used without `return`, it does not have enough knowledge for such a new example. The model thinks each `def` should have at least one `return`. Hence, selected exercise contexts for the training task are also crucial. If the training exercises are too different from the ones the model is deployed for, this could be unfavourable for its performance. Using too simple exercises could also reduce model performance when used to predict misconceptions inside more complex source codes.

```
class EtudiantPEPS(Etudiant):
    def __init__(self, nom, prenom):
        super().__init__(nom, prenom)
```

Listing 6.8: Example predicted `miss_return` even if no `return` is required.

The second limit, i.e. labels' saturation, relates to the model output. Even if an example can have more than four labels, the model never exceeds this number of predicted misconceptions. It even tends to only predict at most three labels per instance. This observation could be another form of mutation over-fitting. Every multi-label example we created for the training task could have at most three labels. The model could have learned this meta-parameter, and its behaviour transcripts this when predicting new instances. Further experiments are required to explore this phenomenon and see if the model over-learned also this aspect.

Conclusion: Even if the model does not demonstrate over-fitting on the training instances during the first experiment, it contains another form of this phenomenon. Rather than learning the instances by heart, it started learning the injected mutations by heart. The model became strong in predicting mutants since the mutations do not change drastically for the instances of the same class. However, when dealing with actual failed submissions, what belongs to a class can come with more variations. Slight changes to the mutations make the model less confident in predicting these cases. Additionally, mutations that could create contradictory examples are misleading the model. These observations emphasize the need for careful mutation designs. For example, by adding new cases to `hardcoded_arg` mutants, the model could have a deeper understanding of this label's meaning. The two following research questions will explore changes to the training process which could mitigate the mutation over-fitting: adding instances or reducing training.

RQ2.3: additional training

Since our model encounters mutation over-fitting, we could provide larger training datasets to reduce this phenomenon's impact. The model could focus on other aspects and mutations inside a more varied set of source codes. We can fulfil this objective by adding more mutants or original submissions into the mutated set.

To explore this aspect, we created four additional datasets. The two firsts contain the 2500 original submissions used for the datasets of the first experiment [6.1.1]. One has 12 mutants per original code instead of 6, while the other keeps all the mutants after duplicate removal. These are called "`more_mutants`", which has 29624 mutants, and "`all_mutants`", which contains 100272 instances. The two other datasets we constructed contain mutants generated from the complete submissions'

set rather than the 2500 randomly selected ones. To limit the generation time, we specified the mutation labelling program to output up to 18 multi-label mutants per original code. The third dataset limits itself to 27977 mutants, while the last keeps the whole generated set, i.e. 95274 mutants. These two are called "`more_codes`" and "`all_codes`". Appendix L presents the statistics table of each of these newly constructed datasets. Each has been split into a train, a validation and a test set.

We launched the training process for the *baseline* configuration three times using the same seeds as for the first experiment. Figure 6.15 shows the loss curves obtained by the first run for each training set, while Table 6.7 shows the performance metrics of each run on their respective test set.

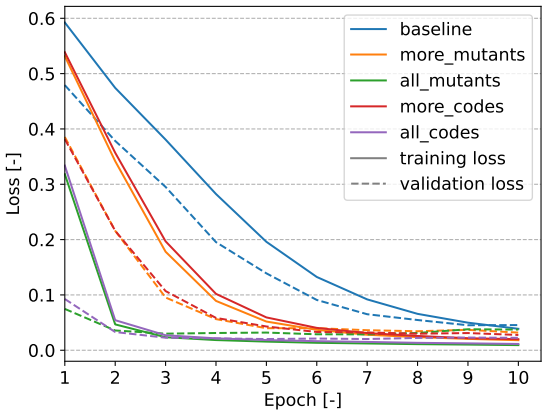


Figure 6.15: Loss curves for *baseline* on the other datasets.

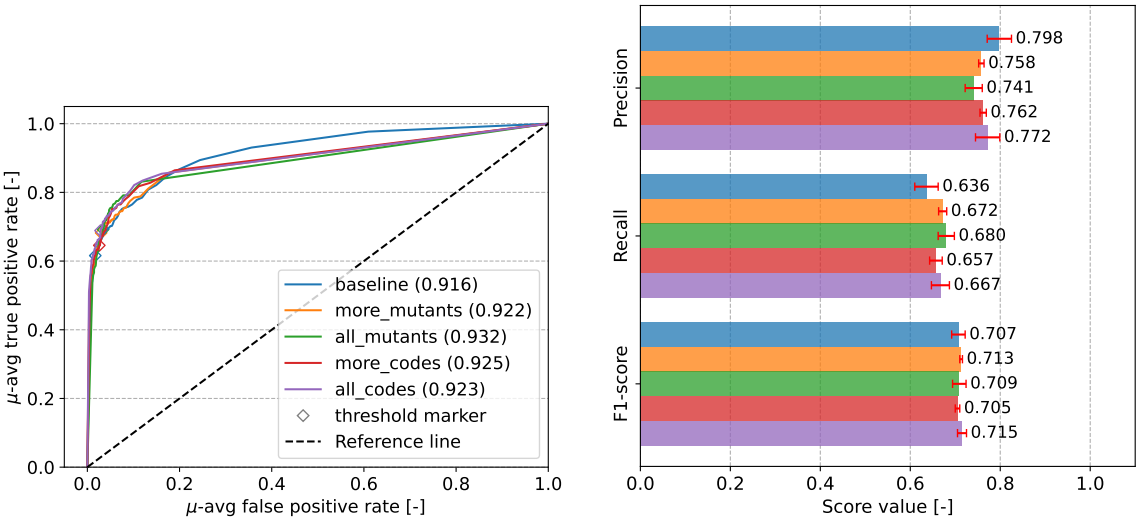
Model	Run 1	Run 2	Run 3
`more_mutants`	0.95360	0.94767	0.94551
`all_mutants`	0.96215	0.95915	0.95938
`more_codes`	0.96287	0.96250	0.96002
`all_codes`	0.97291	0.97254	0.97234

Table 6.7: Performance metrics for the different new datasets. Computed over their respective test set.

As depicted in the figures, using more instances for the training process reduces further the loss. The `all` variants reached their convergence level after three epochs, while the `more` variants reached it around the 7th epoch. However, the lowest loss each model obtains does not exceed 0.05, similar values to the *baseline*.

Reaching these nearly null values could demonstrate that the models understand the mutants better. This observation could also mean that the model over-fitted further the injected mutations. The diversity brought by the added original instances could have been fading out by the model mutation over-fitting. Figures 6.16a and 6.16b show the performance of each newly trained model on the evaluation dataset.

As depicted by the bar plot Figure 6.16b, there is no considerable gap between the average f1-scores each model obtained. The new instances we added during the training helped the model on the training task but did not improve its generalizability on the evaluation dataset. The mutation over-fitting we highlighted



(a) ROC curves for the first seed

(b) Global bar plot (use the same color legend as Figure 6.16a)

Figure 6.16: Evaluation dataset results for the *baseline* trained with larger datasets.

when discussing **RQ2.2** does not disappear when using these larger datasets. The fact that the *baseline* reached equivalent scores to the models using larger datasets means the model learned the mutations correctly and applies its knowledge to the evaluation dataset the same way the other configurations do. The performance limit we could observe is more inherent to the mutation labelling than the training process. Table 6.8 compares the training time (TT) and the obtained scores.

Metrics	*baseline*	`more_mutants`	`all_mutants`	`more_codes`	`all_codes`
Size	15176	29624	100272	27977	95274
Avg TT	$1h\ 11m\ 56$	$2h\ 38m\ 37$	$8h\ 34m\ 08$	$2h\ 16m\ 00$	$7h\ 49m\ 13$
Avg f1-score	0.707	0.713	0.709	0.705	0.715
baseline gain	+0.00%	+0.85%	+0.28%	−0.28%	+1.11%

Table 6.8: Comparison between the average training time (Avg TT) and the average f1-scores. *baseline* gain is the proportion of the *baseline* f1-score each configuration obtained. The training has been performed on one NVidia Tesla V100 GPU.

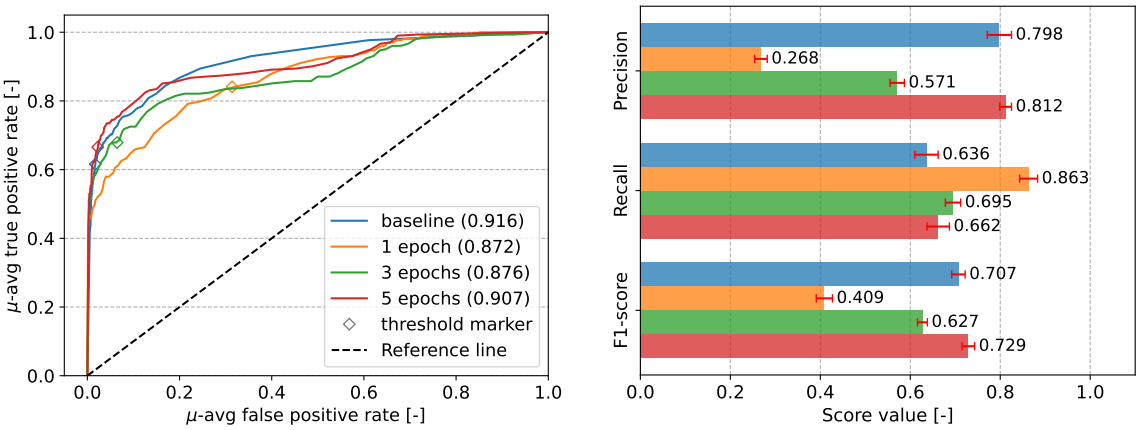
The average gains with respect to the *baseline* depicted in Table 6.8 highlight the limitations of our approach using larger datasets. The training time goes up to 7 times the one of the *baseline* for an incremental gain of less than one per cent. This approach comes with a high resource cost we could spare by using the *baseline*

obtaining comparable performance.

Conclusion: Adding training instances to help the model learn more does not provide better performance on the evaluation dataset. The mutation over-fitting remains, and the model generalizability is not further improved. Furthermore, this strategy comes with a high cost on the training process, using up to 7 times more resources than the *baseline*. Without modifying the mutations we inject to encompass more cases, the instances variety inside the training set does not provide enough power to bypass mutation over-fitting.

RQ2.4: limiting training

The other alternative we explored to limit mutation over-fitting is to train the model for fewer epochs. Doing so has been proven not to significantly impact (up to 2% lost with 5 epochs) the performance of each configuration on the training task. The goal of this section will be to highlight whether these limited training models mitigate the mutation over-fitting. Figures 6.17a and 6.17b show the behaviour of each model on the evaluation dataset.



(a) ROC curves for the first seed

(b) Global bar plot (use the same color legend as Figure 6.17a)

Figure 6.17: Evaluation dataset results for *baseline* with limited number of epochs.

The configurations limited to 1 and 3 epochs obtain f1-scores of 0.409 and 0.627, below the *baseline* performance (0.707). However, the 5 epochs configuration outperformed the *baseline* by obtaining 0.729. This observation means that using half of the *baseline* epoch number gives the model a better understanding of the evaluation instances. We could conclude that the 5 epochs acted like a sweet spot

where the loss has been reduced sufficiently to obtain f1-scores above 0.9 on the training task but not over-fit the mutations as the *baseline*.

However, going from 0.707 to 0.729 is a minor gain, not especially representative of a better understanding of the model. Looking at the label-wise performance in Appendix M, we could recover the same class groups we highlighted when discussing **RQ2.2**. The model lacks comprehension of the `miss_try` and `hardcoded_arg` to mention only these. The performance limits seem not inherent to the model but rather to the representativeness of the mutations we designed.

Conclusion: Reducing the number of epochs does not mitigate entirely the mutation over-fitting. When using less than 3 epochs, the model does not learn enough from the training task and obtains f1-scores lower than 0.65. However, when limited to 5 epochs, the model outperforms the *baseline* by 2.2%. This performance increase does not prove that the model limits mutation over-fitting. Bypassing this phenomenon requires modifying the designed mutations to get closer to specific cases of students' misconceptions, as highlighted by the **RQ2.2** conclusion.

RQ2.5: predictions' interpretation

Due to the black-box nature of our system, we looked at how we could highlight the most contributing part of an input to predict a specific label. To do so, we used the Integrated Gradients attribution scores we presented inside the *Background* chapter [3.5]. Thanks to the `Captum` library, we added tokens highlighting inside the predictions. Appendix E, presenting the Java application that can interface with the model, shows an example of such a highlighting.

The current research question aims to determine whether the interpretability mechanism we added gives better insights into the model predictions and could help students. Unfortunately, due to time limits, we cannot answer this question entirely for now. The attribution procedure is only available with the interactive interfaces we designed, one instance at a time. The evaluation dataset we created contains the windows in which we located each misconception. However, there is currently no mechanism to automatically feed these windows to the model and compute the interpretability coverage of its predictions. This improvement is part of the further works our *Conclusion* chapter will present [7.4].

Nonetheless, we performed a manual experiment on the attribution scores aspect of our model. In specific situations, the output improves understanding of a specific mistake. For example, Listing 6.9 presents the highlights we obtained for a submission of the evaluation dataset. For the `bad_loop` label associated with this instance, the `while` is the most contributing token to predict such a class. This

output would allow the student to identify the erroneous loop.

```
def produits(n):
    t = 0
    for i in range(1,n+1):
        for j in range(1,n+1):
            while i *j == n:
                t = (i,j)
                continue
    return t
```

Listing 6.9: Instance 122 of the evaluation dataset with the highlights of the most contributing token for `bad_loop` (attribution score of 0.95).

However, other cases do not provide such interpretable results and seem to attribute high scores to useless components. This situation is depicted by Listing 6.10 presenting a submission for which the attribution score misleads our interpretation of the results. We would expect the "`t`" to be highlighted, not the function name.

```
def divise_jusque(n):
    for i in range (1,n):
        if not n % i == 0:
            t=i
    return t
```

Listing 6.10: Instance 103 of the evaluation dataset with the highlights of the most contributing tokens for `bad_variable` (attribution scores above 0.25 in absolute value).

The fact that positive scores are associated with parts of the function name made us realize that the variable names could influence the models' predictions. Listing 6.18a shows an example where we modified the function name. One can observe the attribution to have changed and, considering Table 6.18b, that the predicted label scores are also modified. This observation implies that further experiments should be conducted to see the model's robustness regarding variable names. The impact of these components is part of the further works we present in the *Further work* section of this thesis [7.4].

```
def divide_up_to(n):
    for i in range (1,n):
        if not n % i == 0:
            t=i
    return t
```

(a) Input code with the highlight of the most contributing token for `bad_variable` (value of −0.62, other tokens got less than 0.2 in absolute value).

Label	Original	Modified
`correct`	0.20278	0.91336
`failed`	0.17091	0.23192
`bad_variable`	0.86563	0.03018

(b) Score predicted for the original instance (Listing 6.10) and the same function with a different name (Listing 6.18a)

Figure 6.18: Function name change and the output it obtains.

Conclusion: Other aspects could be considered for our current interpretability mechanism. The integral computation is estimated by summation inside the `Captum` library, and one can specify the number of steps for this estimation. Adding more steps, compared to the 10 we used, could provide better results but with a longer computing time. One can also change the input embeddings that are used for the computation. We used the embeddings generated by RoBERTa before the stack of Transformer-encoders (see section 3.3). Exploring these aspects and making them efficient for student feedback are further steps research could tackle.

6.2.4 Conclusion

The *baseline* model we explored in this experiment is our first try to construct a machine learning system to predict student misconceptions. By reaching a precision score of, on average, 0.798 for a recall of 0.636, it could already add information to the current feedback students receive. Nonetheless, the model should be further improved to reach this goal. However, if the system could already support one student, this would be a good step forward to automatize machine learning feedback generation. In the *Conclusion* chapter [7], we will present ideas and research tracks to be conducted for this system.

Regarding this experiment, we validated the training task we designed for the first experiment. Training the model on instances generated by our mutation labelling program provides a correct objective for predicting real students' submissions. Models performing better on the artificial task also perform better on the evaluation dataset we manually created. The conclusions about the parameters' meaning and importance we made during the first experiment hold for real submissions.

The main problem this second experiment highlighted is the mutation over-fitting. The model tends to over-learn the injected modifications, like the `+1` for `bad_range`, the `return` inside `def` or the instructions associated with `hardcoded_arg`. This

other form of over-fitting, on mutations rather than instances, needs to be tackled first for the model to provide better performance. Adding instances without modifying the injections or trying to reduce training does not mitigate this phenomenon significantly. A complete work needs to be dedicated to the mutations' design. This next step could target careful analysis of submissions to identify missed cases or improve existing rules.

Chapter 7

Conclusion

This chapter will provide the concluding words of this thesis. After a brief recap of the results obtained, it will cover the hypotheses this work assumed and the risks for the model's validity. We will also cover the limits of the model based on the observed performance. Finally, further works and research tracks will compose the last discussed section.

7.1 Thesis conclusion

Providing feedback to students is not a trivial task, even for a human rater. Nonetheless, as we stated in our *Introduction*, students need this form of support to account for their shortcomings and strengths. Professors and tutors work on a large range of tasks, and the monitoring of students has to cope with the lack of means or resources for personalized feedback. The tutoring sessions can not be individualized, and the tutor's comments are only global and sporadic.

The Inginious platform already helps numerous students during the semester. The benefits of its deployment are not to be proven anymore. However, designing tests targeting specific misconceptions detection is complex and time-consuming.

These contexts and problems were the origins of this thesis theme. Using the latest advances in machine learning, especially for Natual Language Processing tasks, could relieve the teaching team from the tedious tests' designs. Detecting misconceptions could become automatic and independent of the questions' statement. Providing precise highlights this way could support even better student, while encouraging them to be involved in their learning process.

To reach this goal, we created a classification neural network using code embeddings generated on students' submissions. After covering related work on automatic

feedback generation and code embedding usage, we provided a list of theoretical backgrounds to understand the underlying components of the constructed architecture.

The first step we tackled to create our model was the selection of the embeddings' generator, i.e. the embedder. After experimenting with five possible choices, we identified GraphCodeBERT, a variant of BERT, as the best candidate. Nonetheless, we quickly identified the dataset construction as a limit for training our architecture. We developed a process inspired by DeepBugs by Pradel and Sen [2] called mutation labelling. Thanks to user-defined mutation rules, one could automatically label instances according to specific injections. This mechanism allowed us to create 14 misconceptions classes we wanted our model to predict. Leveraging the automatisation of labelled instances' generation, we obtained extensive datasets for our training processes.

Since we had the embedder, the training instances and the architecture, we needed to check the model behaviour for real cases. Before doing so, a first experiment explored the power of our model. By obtaining f1-scores above 0.9, we highlighted the system to be performing when asked to detect injected misconceptions. This first experiment also enabled analysis of the components of the model. Then, the second experiment explored the model's predictions on students' submissions we labelled manually. We could identify the validity of the training task and what interpretability mechanisms could offer for feedback generation. However, this second experiment highlighted the mutation over-fitting of our system. Rather than overfitting the instances in the training set, the model learned by heart the injected mutations. Changing parameters like the number of epochs or the training set sizes does not provide solutions to this phenomenon. The only ways we identified, for now, would be to include more variations inside the mutations. This limitation could be unsatisfying, but obtaining f1-scores above 0.7 on real students' submissions could be considered a first step towards fully automatic feedback generation.

Going back to the question we asked ourselves at the start of this thesis: "Yes, a machine learning model can play a role in providing feedback to students. The system's addition to the current feedback from Inginious and their tutor could help more students during their learning process. However, our current approach needs improvements to handle more cases and obtain better performance. These points constitute the next steps we could tackle to deploy the system concretely."

7.2 Threats to validity

By taking a step back from our approach, we identified three assumptions we made that could lead to different results if these were modified.

First, there are hypotheses underlying the mutations we designed, like the limited representativeness. Our goal was not to get an extensive list of these, and we ignored some, like, for example, misusing object attributes or "`self`". However, when presenting the misconceptions classes in section 5.2, we assumed that injecting mutations would lead to erroneous codes. Nevertheless, as presented inside our experiments for **RQ1.1** and **RQ2.2**, some mutations, like `miss_parenthesis` or `miss_try`, could create contradictory examples during training. Since such injections do not always lead to incorrect source codes, these cases can mislead the models when faced with actual students' submissions.

We also made hypotheses on the configuration parameters. The fact that the *baseline* configuration obtained f1-scores above 0.9 during the first experiment [6.1] does not imply it would be the best one on the evaluation dataset. When introducing the second experiment [6.2], we assumed each model obtaining performance comparable to the *baseline* to be equally good on the actual students' submissions. However, other configurations could have obtained improved scores on the evaluation dataset. For example, using GELU as classification head activation or disabling batch normalization could improve the generalizability of the model's predictions.

Finally, limits lie inside the evaluation dataset we designed for the second experiment [6.2]. Some instances were complex to classify manually based on the 16 classes we created. Listing 6.5 presented when discussing **RQ2.2** is one example. The submission contains problems on seven lines out of nine. Even if we considered each submission carefully, we could not be sure we detected each error. Furthermore, some of these can be debatable: "Do we consider predicting `bad_range` even if the `range` call is not in a loop?" The current model does not have the flexibility of a human rater. Additionally, the size limit of the evaluation dataset (161 examples for 302 detected mistakes) could bias the results. We could have that the selected instances are closer to contexts the model saw when tackling the training task (cf. first experiment [6.1]). The model could predict these cases with confidence and have more problems with other exercises, like, for example, the Python object question in Listing 6.8.

7.3 Identified model limits

This section discusses the performance limits we identified on the evaluation dataset [6.2].

Adding other exercises inside our selection presented in Appendix C could help the model handle missed cases. For example, adding more questions about file handling could improve the behaviour of the `bad_file` and `miss_try` labels. Adding questions on dictionaries could improve the `bad_list` label.

However, we detected that the model tends to over-learn artificial injections when we answered **RQ2.2**. The model learns them by heart and misses cases that are slight changes to these. Some mutations, like `bad_range` (see Listing 6.4) and `hardcoded_arg` (see Listing 6.5), could be refined for more robustness when predicting actual students' submissions.

Finally, we are not currently using the embedder power at its full potential. GraphCodeBERT has been designed to accept textual input associated with code tokens and DFG. Using this part to contain the question statements could improve the model's performance. Improving this embedder to consider indentation could provide more possible misconceptions to be handled, like misplacing a return inside a loop. Lastly, additions to the DFG representations could help the model understand better students' mistakes.

7.4 Further work

First, we have possible ideas to improve the model's performance:

- limiting mutation over-fitting: improving the mutations and making them more robust to variations from the actual students' misconceptions (see **RQ2.2**),
- including indentation: for example, by adding a special `[IND]` token to represent these or by adding a semicolon at the end of an indented block (cf. section 3.4),
- improving DFG representation: including unused arg and constants inside the graph or by modifying the translation into RoBERTa input (see **RQ1.2**),
- adding textual information: get the questions' statement and add these to the input (cf. 5.3).

Research tracks can also target results analysis:

- comparing pattern mining techniques [2.2] with our model,
- exploring the impact of variable's names: the model can predict differently depending on the function name (see **RQ2.5**),
- deploying the model for other programming languages taught during other University courses,
- improving interpretability by making the model predict a possible token span associated with the set of labels.

Other ideas can continue some preliminary experiments:

- exploring the interpretability of the model: using other methods or parameters and looking at the activation map of the attention layers (see **RQ2.5**)
- determining the environmental impact of the system.

The second point has already been broached to estimate the carbon emissions this thesis created by training all the models for each experiment. We used the `CodeCarbon` tool by Courty et al. [87] to estimate the carbon mass a Python program emits. The computation is based on the energy consumed by the CPU, the RAM and the GPU when running the program. Using the average energy mix of the country in which the model is trained, here Belgium, it can provide information about expected emissions. We performed three runs of the *baseline* from the first experiment [6.1] on the 19th of May 2024 at 00 : 24 am with the CECI infrastructure. Training one run of the *baseline* emits, on average, according to the `CodeCarbon` estimations, 74.662 g of CO_2 equivalent. This value could correspond to approximately 0.34 km of distance travelled by a gasoline-powered car, according to the estimations provided by the United States Environmental Protection Agency[1]. Using this 74 g value as a basis, we could estimate the total CO_2 emission to be 11.374 kg, accounting for each training run presented in this manuscript. This estimation assumes each run to be proportional to the *baseline* according to its number of training instances. This value does not account for all the tests we made on the CECI (we submitted 496 jobs of variable length since the start of this work) and the experiments made on personal computers.

When using the model for prediction on a personal laptop without GPU, we estimate 0.556 mg of CO_2 eq emitted for one predicted instance. Assuming 500 students, using the model during an exercise session with 4 questions and, on average, 5 submissions per student, meaning 10000 requests in total, a tutoring session could

[1] `https://www.epa.gov/energy/greenhouse-gases-equivalencies-calculator-calculations-and-references`

emit 5.56 g of CO_2 eq. This estimation does not include the server emissions and the cost of the requests. Further analyses and considerations need to be tackled to interpret these results and put them into perspective for deployment of the model.

Finally, the last points could be part of further work:

- interfacing the model predictions with Inginious: students would like to receive textual feedback rather than specific labels. They should also be able to use the model-as-a-service rather than running this on their personal computer.
- integrate continuous learning with students' submissions when the model is deployed.

We will address the concluding word of this thesis to the people we mentioned in the Acknowledgments. Thank you for your support, feedback and comments. We hope this work can be continued and finally deployed to help students as much as possible.

Appendix

A RedBaron mutation code example

The following code snippet provides Python function using `RedBaron` to represent a mutation rule.

```python
from redbaron import RedBaron
import random as rd

# If the code raises an error, this is considered as a non matching rule
def remove_assign_while(code: str) -> str:
    # Creates the red structure
    red = RedBaron(code)

    # Get all the while loop nodes
    all_whiles = red.find_all("while")

    # Randomly select a while
    selected_while = rd.choice(all_whiles)

    # Get all the names used inside the condition
    test_vars = selected_while.test.find_all("name")

    # Construct a set of names in the condition
    set_vars = {t.value for t in test_vars}

    # Find all the assign inside the body of the loop
    all_assign = selected_while.value.find_all("assign")

    # Keep only the assign that updates the names in the condition
```

```
    kept_assign = [a for a in all_assign if a.target.value in set_vars]

    # Remove these assign nodes
    for k in kept_assign:
        k.parent.remove(k)

    # Return the mutated code
    return red.dumps()
```

B Designed RedBaron rules

This appendix covers the RedBaron rules we designed for our mutation labelling program, each with a short explanation. For more information about how to use these rules and their meaning, you can check their corresponding codes and information on GitHub[2]. We only used part of these to create the 15 classes we present in Appendix D.

1. `remove_try`: Replace a complete try-except block with the content of the try.
2. `other_variable`: Replace a variable with another variable inside the same code snippet.
3. `random_variable`: Replace a variable with a random variable not already inside the code snippet.
4. `value_change (TYPE) (VALUE)`: Select a value of type TYPE and add VALUE to it
5. `remove_assign_while (INIT)`: Remove all the assignments where the left-hand side corresponds to an expression in the condition of a while loop.
6. `with_to_open`: Replace a `with`-block with the content of its body prepended with the assignment in the `with` statement.
7. `remove_return`: Remove a return statement from the code snippet.
8. `for_range`: Replace a "`for VAR in VAL`" with "`for VAR in range(VAL)`" if VAL does not contain a reference to range.
9. `revert_slice`: Exchange the lower and upped bound value in a slicing operation.
10. `suppress_part_slice`: Remove a value inside a slicing operation.
11. `remove_arg_call`: Remove an argument from a call to a function.
12. `remove_call`: Completely remove a call to a function.
13. `change_sign (TYPE)`: Change the sign of a value of type TYPE.
14. `remove_nested_loop`: Replace a loop with its body.

[2] `https://github.com/StevenGuyCap/CESReS-model/tree/main/mutation`

15. `remove_if_else`: Replace an if-else block with the content of the if body.
16. `suppress_if`: Remove a complete if-else block.
17. `list_to_tuple`: Replace a list instantiation (values enclosed in brackets) with a tuple creation.
18. `change_comparison`: Randomly replace a binary/equality comparison with another operator.
19. `identity`: Return the original code.
20. `remove_self`: Remove a variable named `self` anywhere inside the input code.
21. `misplace_return`: Move a return instruction to another place inside the body of the function containing this instruction.
22. `assign_to_comparison`: Replace an assign operation (`=`) with an equality comparison operator (`==`).
23. `comparison_to_assign`: Replace an equality comparison operator (`==`) with an assign operation (`=`).
24. `change_operand_order`: Exchange the operands inside a binary operation.
25. `clear_aug_assign`: Remove the operator inside an augmented assign.
26. `unravel_aug_assign`: Replace an augmented assign with its complete formulation.
27. `ravel_assign`: Replace an expression using a binary operator to update a value with an augmented assign equivalent.
28. `return_in_init`: Find a definition to the method `__init__` and add, as the last body instruction, a return for a dummy value (either self or the class name).
29. `out_of_bounds_range`: Add a `+1` inside the expression found inside a random call to the `range` function.
30. `division_change`: Replace `/` with `//` and conversely.
31. `print_before_return`: Add a print instruction as an instruction preceding a return statement.

32. `indent_return`: Select a return statement appearing just after an indented block and add one level of indentation to the selected instruction.

33. `bad_open_mode`: Replace the opening mode inside the arguments of an `open` function call.

34. `miss_close`: Supress a call to methods named `close`.

35. `miss_except`: Remove an except block inside a try-except construction.

36. `replace_all_var`: Randomly rename a variable and all its occurrences.

37. `rename_all_def`: Rename all functions defined in the code snippet to have a dummy name.

38. `rename_all_vars_dummy`: Rename all variables defined in the code snippet to have a dummy name.

39. `remove_comments_and_docstrings`: Remove all the comments and the docstrings.

40. `remove_parenthesis`: Replace an associative parenthesis expression with the content enclosed in the parentheses.

41. `hardcode_arg`: Add an instruction at the start of the body of a function with at least one argument where one of the function arguments is assigned to a random integer between 0 and 20.

C Selected semester exercises

This section will provide the names of the semester questions available on Inginious we selected to create our datasets. The associated course is available online on the syllabus[3] as well as on the Inginious platform maintained by the Université Catholique de Louvain[4]. The exam submissions we used during the second experiment will not be made publicly available because of privacy concerns about the students' submissions.

The following list covers the exercises we selected:

1. Session 3: Q* PGCD
2. Session 3: Q* Fibonacci
3. Session 3: Q* Indice BMI
4. Session 3: Q* Amende pour excès de vitesse
5. Session 4: Triangle
6. Session 4: Carre
7. Session 4: Index maximum
8. Session 4: Recherche
9. Session 4 : Nombres premiers
10. Session 4: Q* Hogwarts - Fat Lady
11. Session 4: Q* Anonymous - Information extraction
12. Session 4: Q* Anonymous - Information treatment
13. Session 5: Compteur d'événement
14. Session 5: Recherche binaire
15. Session 6: Q* Représentation de tableau
16. Session 6: Q* Hogwarts - Admission letter
17. Session 6: Q* Hogwarts - Quidditch

[3] `https://syllabus-interactif.info.ucl.ac.be/index/info1-exercises`
[4] `https://inginious.info.ucl.ac.be/course/LSINF1101-PYTHON`

18. Session 6: QBF
19. Session 7: Egalité entre structures
20. Session 7: Création de dictionnaire pour une liste de mots
21. Session 7: Création de dictionnaire
22. Session 7: Création de dictionnaire des valeurs maximums
23. Session 7: Utilisation des clés

D Selected misconceptions

This appendix presents the list of misconceptions and mistakes we designed for injection inside the submissions collected from Inginious. We will highlight the reasons for and the hypotheses behind the choices we made to design these. Some injected errors are based on what we could observe from students' bad habits or actual erroneous submissions. Other mutations found were inspired by the work of Chiodini et al. [85].

`correct`

This class is the simplest one. It contains every mutant that does not have a different behaviour than the original submission. The rule set associated with this label contains five mutations:

- `print_before_return`: adds a print before a return statement.
- `change_operand_order`: exchange two operands from a binary operation.
- `unravel_operation_assign`: replace an augmented assign with its classical formulation.
- `ravel_assign`: replace a binary operation with an augmented assign equivalent.
- `identity`: return the original input.

These rules do not modify what the source code aims to compute. We created this rule set to make the model understand more the other injected mistakes. For example, having a print inside a function is not a mistake if every other part is present. However, without adding some prints inside the "correct" instances, the model would directly classify codes with call to `print` as `print_return`. The cases generated by this rule set add diversity to the training instances.

`print_return`

This class represents students who misunderstood the use of `print`. This label assumes that successful submissions should return something after executing. The corresponding mutation replaces all the return statements with equivalent prints. Listings 8.1a and 8.1b show a mutation example.

```
def is_a_greater_than_b(a,b):
    if a > b:
        return a
    else:
        return b
```

(a) Original code.

```
def is_a_greater_than_b(a,b):
    if a > b:
        print(a)
    else:
        print(b)
```

(b) Mutated code.

Listing 8.1: Mutation example for the `print_return` label.

`miss_return`

This class should include students who have problems with the return statement. Either they forgot when this statement is required or what is its goal. This label assumes that successful submissions' functions should return something after executing. Codes are mutated by removing a return randomly from the code snippets. This label also appears when all the returns are replaced with prints (see `print_return`). Listings 8.2a and 8.2b show a mutation example.

```
def condition_sum(a,b):
    if a > b:
        out = a + b
        return out
    else:
        return b
```

(a) Original code.

```
def condition_sum(a,b):
    if a > b:
        out = a + b
    else:
        return b
```

(b) Mutated code.

Listing 8.2: Mutation example for the `miss_return` label.

`bad_division`

This label represents misconceptions linked to `/` and `//`. Some students do not understand which to use to fulfill a specific task. The corresponding mutation randomly replaces one of the two operators with the other. Listings 8.3a and 8.3b show a mutation example.

```
def quotient_reminder(a,b):
    quo = a // b
    rem = a % b
    return quo, rem
```

(a) Original code.

```
def quotient_reminder(a,b):
    quo = a / b
    rem = a % b
    return quo, rem
```

(b) Mutated code.

Listing 8.3: Mutation example for the `bad_division` label.

bad_variable

Some students write code thinking that specific variables they have in mind are also available when executing their code. Others are using a variable that should not be part of a specific instruction. This class encompasses these two cases, by using `other_variable` and `random_variable` rules (see Appendix B). Listings 8.4a and 8.4b show a mutation example where a variable is replaced randomly with another (`other_variable`).

```
def quotient_reminder(a,b):
    quo = a // b
    rem = a % b
    return quo, rem
```

(a) Original code.

```
def quotient_reminder(a,b):
    quo = a // a
    rem = a % b
    return quo, rem
```

(b) One possible mutated code.

Listing 8.4: Mutation example for the `bad_variable` label.

bad_loop

When using loops inside their submissions some students misunderstand when they should use a `while` or a `for`. They sometimes also use `while`s as if these blocks were updating the condition automatically, creating infinite loops. This situation also arises when they use "True" as a condition. Others are confused between `if`s and `while`s. This label should gather all these misconceptions. Listings 8.5a and 8.5b show a mutation example where the update statement is removed from the loop (`remove_assign_while`).

```python
def sum_from_a_to_b(a,b):
    out = 0
    i = a
    while i < b:
        out += i
        i += 1
    return out
```

(a) Original code.

```python
def sum_from_a_to_b(a,b):
    out = 0
    i = a
    while i < b:
        out += i
    return out
```

(b) Mutated code.

Listing 8.5: Mutation example for the `bad_loop` label.

`miss_loop`

This class corresponds to students who forgot to use a loop inside their submission. By using `remove_nested_loop`, the related mutation replaces a loop with its body. Listings 8.6a and 8.6b show a mutation example.

```python
def sum_from_a_to_b(a,b):
    out = 0
    i = a
    while i < b:
        out += i
        i += 1
    return out
```

(a) Original code.

```python
def sum_from_a_to_b(a,b):
    out = 0
    i = a
    out += i
    i += 1
    return out
```

(b) Mutated code.

Listing 8.6: Mutation example for the `miss_loop` label.

`bad_range`

Students who misunderstand the range function should be grouped within this class. The mutation set will change how the `range` function is used: either by adding `+1` inside the call or by removing this function call. Listings 8.7a and 8.7b show a mutation example for the second case.

```python
def sum_to(b):
    out = 0
    for i in range(b):
        out += i
    return out
```

(a) Original code.

```python
def sum_to(b):
    out = 0
    for i in b:
        # or range(b+1)
        out += i
    return out
```

(b) Mutated code. The comment depicts another possible injection.

Listing 8.7: Mutation example for the `bad_range` label.

bad_assign

This class corresponds to students who are confused about the use of `=` and `==`. They are misusing one for the other. To create the instances of this class, the corresponding mutation rule set replaces one of the two operators with the other. Listings 8.8a and 8.8b show a mutation example.

```python
def sum_from_a_to_b(a,b):
    out = 0
    if a == b:
        return out
    for i in range(a, b):
        out += i
    return out
```

(a) Original code.

```python
def sum_from_a_to_b(a,b):
    out = 0
    if a = b:
        return out
    for i in range(a, b):
        out += i
    return out
```

(b) Mutated code.

Listing 8.8: Mutation example for the `bad_assign` label.

bad_list

Students that could be helped by this class detection are the ones who misused lists. They could think that "`append`" returns a new list instead of having a side effect or use tuples as if they were lists. This label also includes misconceptions where students put values at indexes that are not initialized yet. Listings 8.9a and 8.9b show a mutation example.

```python
def list_from_a_to_b(a,b):
    out = []
    for i in range(a, b):
        out.append(i)
    return out

```

(a) Original code.

```python
def list_from_a_to_b(a,b):
    out = [] # or ()
    for i in range(a, b):
        out = out.append(i)
        # or out[0] = i
    return out
```

(b) Mutated code. The comments depict other possible injections.

Listing 8.9: Mutation example for the `bad_list` label.

bad_file

The `bad_file` class encompasses the problems students could have when dealing with files. They should remember to close their files if they do not use a `with` statement and open these with the correct permissions. The mutations corresponding to this label either replace a `with` with an `open` without `close` or change the opening permissions. Listings 8.10a and 8.10b show a mutation example.

```python
def get_lines(file):
    with open(file, 'r') as f:
        lines = f.readlines()
    return lines
```

(a) Original code.

```python
def get_lines(file):
    f = open(file, 'r')
    lines = f.readlines()
    return lines
```

(b) Mutated code.

Listing 8.10: Mutation example for the `bad_file` label.

miss_try

When handling files, students should remember that an Exception could be thrown if the files do not exist. They should also be sure that there are enough except clauses to catch the possible errors. This label will either replace a try-expect block with its try-body or remove an except clause. Listings 8.11a and 8.11b show a mutation example.

```python
def get_lines(file):
    try:
        with open(file) as f:
            lines = f.readlines()
        return lines
    except:
        return []
```

(a) Original code.

```python
def get_lines(file):
    with open(file) as f:
        lines = f.readlines()
    return lines
```

(b) Mutated code.

Listing 8.11: Mutation example for the `miss_try` label.

`miss_parenthesis`

This label assumes that students who use associative parentheses are only using these when needed. Forgetting these would lead to incorrect mathematical expressions and erroneous answers. The corresponding mutation randomly removes an associative parenthesis block. Listings 8.12a and 8.12b show a mutation example.

```python
def compute_mean(a, b):
    out = (a+b)/2
    return out
```

(a) Original code.

```python
def compute_mean(a, b):
    out = a+b/2
    return out
```

(b) Mutated code.

Listing 8.12: Mutation example for the `miss_parenthesis` label.

`hardcoded_arg`

Some students think variables, especially the function arguments, should be hard-coded inside the code. This label covers these cases by mutating code to add a single assignment instruction to one random arg of a function as the first body instruction. The value assigned to the argument will always be a random integer between 0 and 20. Listings 8.13a and 8.13b show a mutation example.

```python
def sum_to(b):
    out = 0
    for i in range(b):
        out += i
    return out

```

(a) Original code.

```python
def sum_to(b):
    b = 18
    out = 0
    for i in range(b):
        out += i
    return out
```

(b) Mutated code. 18 is a random number between 0 and 20.

Listing 8.13: Mutation example for the `hardcoded_arg` label.

`overwrite_not_increment`

Finally, this last class covers the problems students could have when using augmented assignments. Some students think that using `=` is sufficient to increment a variable which would remember its previous value. The mutation associated with this class removes an augmented assign operator. Listings 8.14a and 8.14b show a mutation example.

```python
def sum_to(b):
    out = 0
    for i in range(b):
        out += i
    return out
```

(a) Original code.

```python
def sum_to(b):
    out = 0
    for i in range(b):
        out = i
    return out
```

(b) Mutated code.

Listing 8.14: Mutation example for the `overwrite_not_increment` label.

E GUI interface

The GUI called `CodeLabellingGUI` was initially designed to allow one to manually label code snippets. This program is a Java application built with Swing and AWT. A compiled version of the first snapshot of this program is available on this thesis GitHub repository[5]. However, in this version, bugs are still possible.We provide this program without guarantee. Figure 8.15 shows its appearance.

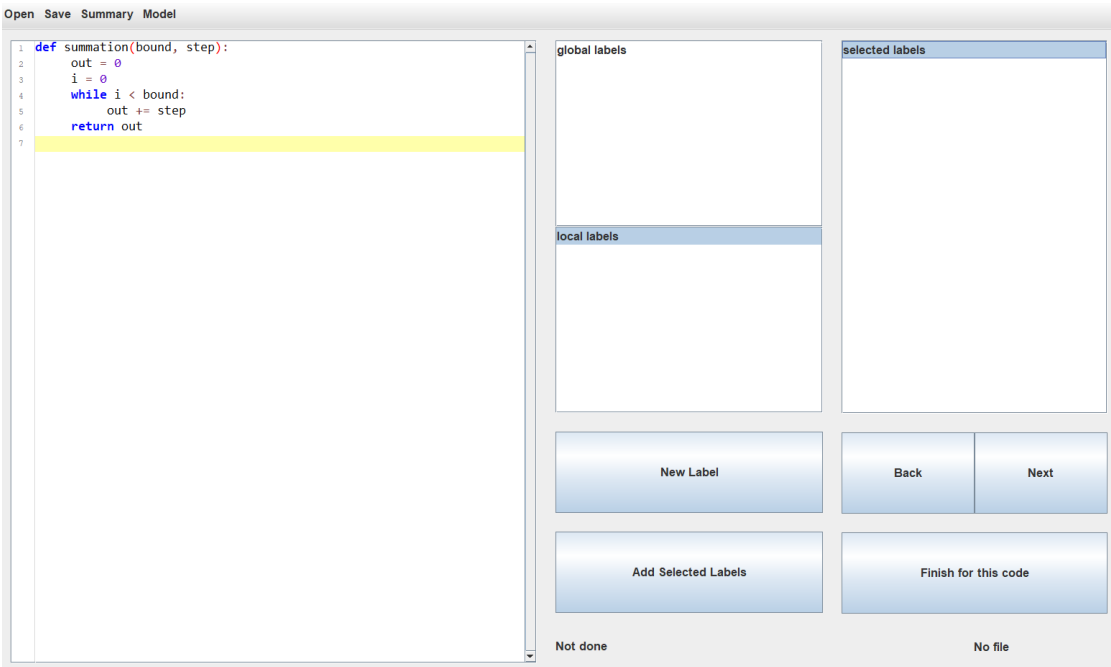


Figure 8.15: `CodeLabellingGUI` program in labelling mode.

The code snippets are displayed on the left, and the user can see possible labels inside the boxes in the middle. The labels already associated with the code snippet are displayed in the list on the right.

We wanted then to reuse this program, without losing its initial functionality, to get the model predictions. By using the `Model` menu, the user can switch on the connectivity with the model. The GUI is now used for prediction, no more for labelling. Figure 8.16 presents this alternative mode.

[5] `https://github.com/StevenGuyCap/CESReS-model/tree/main/gui`

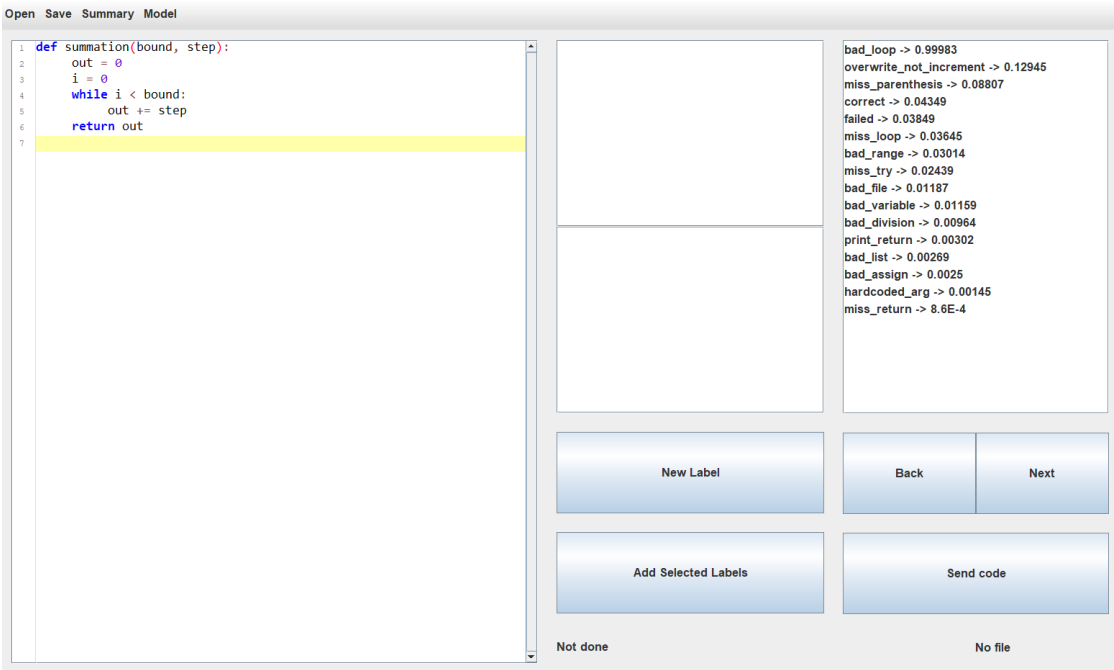


Figure 8.16: `CodeLabellingGUI` program in model mode.

The code section on the left can be modified by the user. When clicking on the `Send code` button, the provided snippet will be sent to the model which should run with `gui` set to true in its configuration.

Once the model has finished its prediction, the GUI displays the labels in the list on the right. Clicking once on a label sends a request to the model for the interpretability. Figure 8.17 shows the results one could obtain.

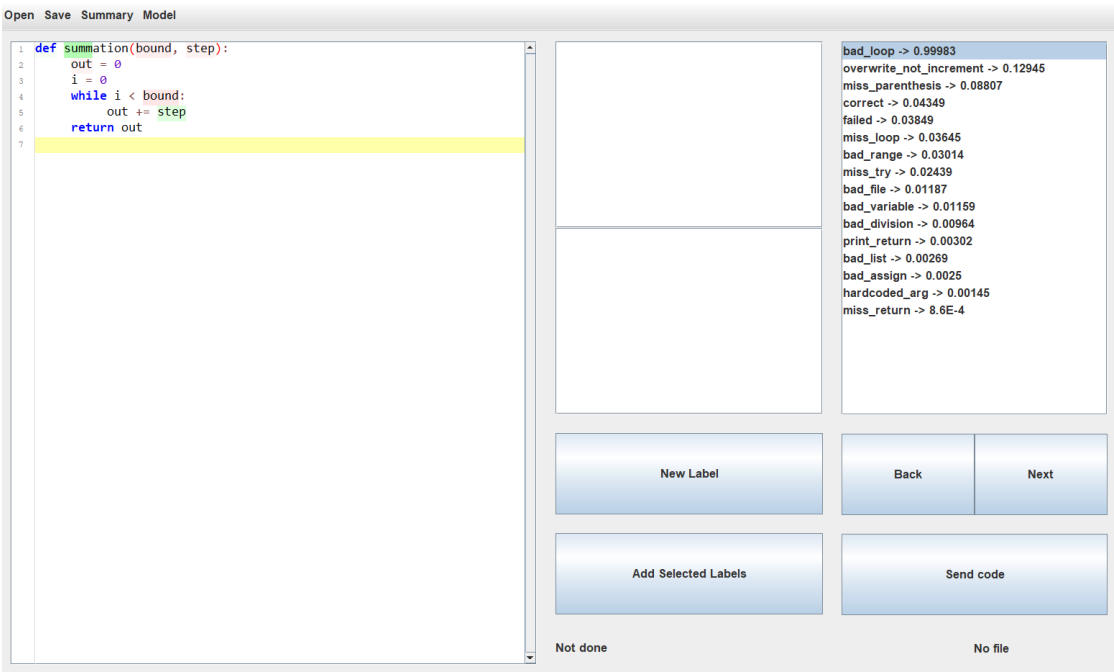


Figure 8.17: `CodeLabellingGUI` program with interpretability output.

The darker the green is, the more its attribution score is close to 1. Darker shades of red mean scores get close to -1.

On our GitHub repository, we provide a working configuration file with a script to launch the GUI and the model simultaneously. The provided readme explains how to connect the two independent parts.

F Reference dataset statistics

This appendix presents the number of instances associated with each label in the dataset used in the first experiment [6.1]. The name `o-n-i` is the abbreviation of `overwrite_not_increment` and `miss-par` of `miss_parenthesis`. The first table shows the proportions for groups of labels.

Labels group	Train	Validation	Test	Total
- `bad_loop`	392 (3.189%)	53 (3.909%)	41 (2.687%)	486 (3.202%)
- `bad_variable`	367 (2.985%)	33 (2.434%)	46 (3.014%)	446 (2.939%)
- `hardcoded_arg`	355 (2.888%)	41 (3.024%)	41 (2.687%)	437 (2.880%)
- `bad_loop` - `miss_return` - `print_return`	347 (2.823%)	44 (3.245%)	39 (2.556%)	430 (2.833%)
- `hardcoded_arg` - `miss_return` - `print_return`	339 (2.757%)	36 (2.655%)	50 (3.277%)	425 (2.800%)
- `bad_assign`	340 (2.766%)	34 (2.507%)	30 (1.966%)	404 (2.662%)
- `miss_return` - `print_return`	331 (2.692%)	29 (2.139%)	42 (2.752%)	402 (2.649%)
- `hardcoded_arg` - `miss_return`	325 (2.644%)	35 (2.581%)	36 (2.359%)	396 (2.609%)
- `bad_variable` - `miss_return`	313 (2.546%)	34 (2.507%)	48 (3.145%)	395 (2.603%)
- `bad_variable` - `hardcoded_arg`	307 (2.497%)	41 (3.024%)	42 (2.752%)	390 (2.570%)
- `bad_assign` - `miss_return` - `print_return`	308 (2.505%)	36 (2.655%)	40 (2.621%)	384 (2.530%)
- `bad_variable` - `miss_return` - `print_return`	303 (2.465%)	33 (2.434%)	44 (2.883%)	380 (2.504%)

- miss_return	307 (2.497%)	32 (2.360%)	38 (2.490%)	377 (2.484%)
- failed	293 (2.383%)	34 (2.507%)	37 (2.425%)	364 (2.399%)
- bad_assign - bad_variable	268 (2.180%)	35 (2.581%)	47 (3.080%)	350 (2.306%)
- bad_assign - hardcoded_arg	275 (2.237%)	29 (2.139%)	33 (2.163%)	337 (2.221%)
- bad_loop - bad_variable	260 (2.115%)	24 (1.770%)	41 (2.687%)	325 (2.142%)
- bad_loop - hardcoded_arg	260 (2.115%)	27 (1.991%)	26 (1.704%)	313 (2.062%)
- bad_variable - miss_loop	259 (2.107%)	24 (1.770%)	29 (1.900%)	312 (2.056%)
- miss_loop	255 (2.074%)	25 (1.844%)	27 (1.769%)	307 (2.023%)
- bad_loop - miss_return	244 (1.985%)	19 (1.401%)	41 (2.687%)	304 (2.003%)
- hardcoded_arg - miss_loop	248 (2.017%)	32 (2.360%)	23 (1.507%)	303 (1.997%)
- correct	246 (2.001%)	26 (1.917%)	27 (1.769%)	299 (1.970%)
- miss_loop - miss_return	236 (1.920%)	28 (2.065%)	31 (2.031%)	295 (1.944%)
- bad_range	234 (1.903%)	36 (2.655%)	23 (1.507%)	293 (1.931%)
- bad_assign - miss_return	218 (1.773%)	35 (2.581%)	27 (1.769%)	280 (1.845%)
- bad_range - bad_variable	215 (1.749%)	30 (2.212%)	27 (1.769%)	272 (1.792%)
- miss_loop - miss_return - print_return	213 (1.733%)	27 (1.991%)	29 (1.900%)	269 (1.773%)
- bad_range - miss_return	214 (1.741%)	26 (1.917%)	28 (1.835%)	268 (1.766%)
- bad_loop				

- miss_loop	216 (1.757%)	24 (1.770%)	25 (1.638%)	265 (1.746%)
- bad_assign - bad_loop	219 (1.781%)	16 (1.180%)	29 (1.900%)	264 (1.740%)
- bad_assign - miss_loop	200 (1.627%)	20 (1.475%)	24 (1.573%)	244 (1.608%)
- bad_range - miss_return - print_return	196 (1.594%)	24 (1.770%)	20 (1.311%)	240 (1.581%)
- bad_range - hardcoded_arg	188 (1.529%)	28 (2.065%)	23 (1.507%)	239 (1.575%)
- bad_assign - bad_range	184 (1.497%)	19 (1.401%)	18 (1.180%)	221 (1.456%)
- bad_loop - bad_range	174 (1.415%)	15 (1.106%)	27 (1.769%)	216 (1.423%)
- bad_range - miss_loop	138 (1.122%)	14 (1.032%)	22 (1.442%)	174 (1.147%)
- bad_list	125 (1.017%)	9 (0.664%)	12 (0.786%)	146 (0.962%)
- bad_list - miss_loop	103 (0.838%)	10 (0.737%)	13 (0.852%)	126 (0.830%)
- bad_list - bad_variable	102 (0.830%)	12 (0.885%)	12 (0.786%)	126 (0.830%)
- bad_list - miss_return	104 (0.846%)	8 (0.590%)	9 (0.590%)	121 (0.797%)
- bad_list - bad_loop	92 (0.748%)	9 (0.664%)	13 (0.852%)	114 (0.751%)
- bad_assign - bad_list	87 (0.708%)	15 (1.106%)	10 (0.655%)	112 (0.738%)
- bad_list - hardcoded_arg	90 (0.732%)	11 (0.811%)	9 (0.590%)	110 (0.725%)
- bad_list - bad_range	92 (0.748%)	10 (0.737%)	7 (0.459%)	109 (0.718%)

- bad_file	95 (0.773%)	8 (0.590%)	4 (0.262%)	107 (0.705%)
- bad_list				
- miss_return				
- print_return	76 (0.618%)	10 (0.737%)	6 (0.393%)	92 (0.606%)
- hardcoded_arg				
- miss_parenthesis	67 (0.545%)	6 (0.442%)	12 (0.786%)	85 (0.560%)
- bad_variable				
- miss_parenthesis	70 (0.569%)	4 (0.295%)	11 (0.721%)	85 (0.560%)
- miss_parenthesis	67 (0.545%)	4 (0.295%)	8 (0.524%)	79 (0.521%)
- miss_loop				
- o-n-i	60 (0.488%)	8 (0.590%)	7 (0.459%)	75 (0.494%)
- hardcoded_arg				
- o-n-i	61 (0.496%)	5 (0.369%)	6 (0.393%)	72 (0.474%)
- miss_return				
- o-n-i	54 (0.439%)	8 (0.590%)	8 (0.524%)	70 (0.461%)
- miss_return				
- o-n-i				
- print_return	59 (0.480%)	4 (0.295%)	6 (0.393%)	69 (0.455%)
- bad_range				
- o-n-i	55 (0.447%)	3 (0.221%)	11 (0.721%)	69 (0.455%)
- bad_variable				
- o-n-i	56 (0.456%)	8 (0.590%)	5 (0.328%)	69 (0.455%)
- miss_parenthesis				
- miss_return				
- print_return	56 (0.456%)	4 (0.295%)	7 (0.459%)	67 (0.441%)
- o-n-i	53 (0.431%)	5 (0.369%)	9 (0.590%)	67 (0.441%)
- bad_loop				
- o-n-i	49 (0.399%)	6 (0.442%)	7 (0.459%)	62 (0.409%)
- bad_file				
- bad_loop	52 (0.423%)	6 (0.442%)	2 (0.131%)	60 (0.395%)
- bad_assign				

- o-n-i	44 (0.358%)	3 (0.221%)	10 (0.655%)	57 (0.376%)
- bad_loop - miss_parenthesis	45 (0.366%)	9 (0.664%)	2 (0.131%)	56 (0.369%)
- bad_assign - miss_parenthesis	41 (0.333%)	5 (0.369%)	10 (0.655%)	56 (0.369%)
- bad_division	44 (0.358%)	4 (0.295%)	3 (0.197%)	51 (0.336%)
- bad_division - hardcoded_arg	44 (0.358%)	1 (0.074%)	4 (0.262%)	49 (0.323%)
- miss_loop - miss_parenthesis	32 (0.260%)	7 (0.516%)	8 (0.524%)	47 (0.310%)
- miss_parenthesis - miss_return	38 (0.309%)	2 (0.147%)	5 (0.328%)	45 (0.297%)
- bad_list - miss_parenthesis	30 (0.244%)	4 (0.295%)	3 (0.197%)	37 (0.244%)
- bad_division - miss_parenthesis	28 (0.228%)	4 (0.295%)	3 (0.197%)	35 (0.231%)
- bad_assign - bad_division	26 (0.211%)	1 (0.074%)	7 (0.459%)	34 (0.224%)
- bad_division - bad_variable	27 (0.220%)	3 (0.221%)	3 (0.197%)	33 (0.217%)
- bad_division - miss_return	23 (0.187%)	6 (0.442%)	3 (0.197%)	32 (0.211%)
- bad_division - miss_return - print_return	26 (0.211%)	1 (0.074%)	5 (0.328%)	32 (0.211%)
- bad_range - miss_parenthesis	22 (0.179%)	3 (0.221%)	5 (0.328%)	30 (0.198%)
- bad_division - bad_loop	23 (0.187%)	3 (0.221%)	3 (0.197%)	29 (0.191%)
- bad_file - bad_range	25 (0.203%)	0 (0.000%)	4 (0.262%)	29 (0.191%)

- bad_list				
- o-n-i	21 (0.171%)	2 (0.147%)	3 (0.197%)	26 (0.171%)
- bad_assign				
- bad_file	18 (0.146%)	2 (0.147%)	1 (0.066%)	21 (0.138%)
- bad_file				
- hardcoded_arg	19 (0.155%)	1 (0.074%)	0 (0.000%)	20 (0.132%)
- miss_try	15 (0.122%)	2 (0.147%)	2 (0.131%)	19 (0.125%)
- bad_file				
- bad_variable	18 (0.146%)	0 (0.000%)	1 (0.066%)	19 (0.125%)
- bad_loop				
- miss_try	15 (0.122%)	2 (0.147%)	0 (0.000%)	17 (0.112%)
- bad_file				
- miss_loop	12 (0.098%)	2 (0.147%)	3 (0.197%)	17 (0.112%)
- hardcoded_arg				
- miss_try	13 (0.106%)	2 (0.147%)	2 (0.131%)	17 (0.112%)
- bad_file				
- miss_return				
- print_return	13 (0.106%)	3 (0.221%)	1 (0.066%)	17 (0.112%)
- bad_division				
- miss_loop	13 (0.106%)	1 (0.074%)	3 (0.197%)	17 (0.112%)
- bad_division				
- bad_list	10 (0.081%)	1 (0.074%)	5 (0.328%)	16 (0.105%)
- bad_file				
- o-n-i	12 (0.098%)	2 (0.147%)	1 (0.066%)	15 (0.099%)
- miss_return				
- miss_try				
- print_return	13 (0.106%)	1 (0.074%)	1 (0.066%)	15 (0.099%)
- bad_file				
- miss_return	10 (0.081%)	0 (0.000%)	2 (0.131%)	12 (0.079%)
- bad_assign				
- miss_try	7 (0.057%)	1 (0.074%)	2 (0.131%)	10 (0.066%)

- miss_return				
- miss_try	8 (0.065%)	1 (0.074%)	0 (0.000%)	9 (0.059%)
- bad_file				
- bad_list	7 (0.057%)	0 (0.000%)	2 (0.131%)	9 (0.059%)
- bad_range				
- miss_try	6 (0.049%)	2 (0.147%)	1 (0.066%)	9 (0.059%)
- miss_loop				
- miss_try	6 (0.049%)	0 (0.000%)	2 (0.131%)	8 (0.053%)
- bad_file				
- miss_try	7 (0.057%)	1 (0.074%)	0 (0.000%)	8 (0.053%)
- bad_variable				
- miss_try	7 (0.057%)	0 (0.000%)	0 (0.000%)	7 (0.046%)
- miss_parenthesis				
- o-n-i	4 (0.033%)	2 (0.147%)	1 (0.066%)	7 (0.046%)
- bad_file				
- miss_parenthesis	5 (0.041%)	0 (0.000%)	0 (0.000%)	5 (0.033%)
- bad_list				
- miss_try	3 (0.024%)	0 (0.000%)	0 (0.000%)	3 (0.020%)
- miss_try				
- o-n-i	2 (0.016%)	1 (0.074%)	0 (0.000%)	3 (0.020%)
TOTAL:	12294	1356	1526	15176

This second table shows the proportions of each label. The percentages represent the proportion of instances in which the label appears.

Labels	Train	Validation	Test	Total
- `miss_return`	4374 (35.578%)	486 (35.841%)	566 (37.090%)	5426 (35.754%)
- `bad_variable`	2572 (20.921%)	281 (20.723%)	356 (23.329%)	3209 (21.145%)
- `hardcoded_arg`	2591 (21.075%)	295 (21.755%)	307 (20.118%)	3193 (21.040%)
- `bad_loop`	2388 (19.424%)	257 (18.953%)	296 (19.397%)	2941 (19.379%)
- `print_return`	2280 (18.546%)	252 (18.584%)	290 (19.004%)	2822 (18.595%)
- `bad_assign`	2235 (18.180%)	251 (18.510%)	288 (18.873%)	2774 (18.279%)
- `miss_loop`	1991 (16.195%)	222 (16.372%)	246 (16.121%)	2459 (16.203%)
- `bad_range`	1743 (14.178%)	210 (15.487%)	216 (14.155%)	2169 (14.292%)
- `bad_list`	942 (7.662%)	101 (7.448%)	104 (6.815%)	1147 (7.558%)
- `o-n-i`	530 (4.311%)	57 (4.204%)	74 (4.849%)	661 (4.356%)
- `miss-par`	505 (4.108%)	54 (3.982%)	75 (4.915%)	634 (4.178%)
- `failed`	293 (2.383%)	34 (2.507%)	37 (2.425%)	364 (2.399%)
- `bad_file`	293 (2.383%)	25 (1.844%)	21 (1.376%)	339 (2.234%)
- `bad_division`	264 (2.147%)	25 (1.844%)	39 (2.556%)	328 (2.161%)
- `correct`	246 (2.001%)	26 (1.917%)	27 (1.769%)	299 (1.970%)
- `miss_try`	102 (0.830%)	13 (0.959%)	10 (0.655%)	125 (0.824%)

Table 8.2: Statistics for the datasets of the first experiment [6.1].

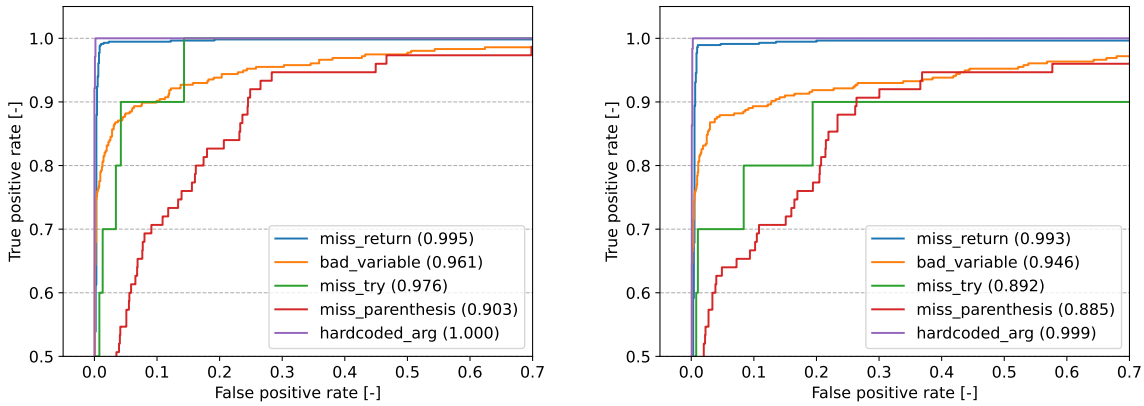
G All results of the "Model parameters" experiment

Model part	Configuration	Run 1	Run 2	Run 3
	baseline	0.93915	0.93733	0.93499
Input generation	`[0, 512]`	0.32725	0.34241	0.34548
	`[128, 384]`	0.93915	0.94010	0.93499
	`[384, 128]`	0.93915	0.94010	0.93499
	`[512, 0]`	0.94360	0.94043	0.94188
Embedder	No fine-tuning	0.60205	0.60382	0.60050
Classification head	0 layer	0.93092	0.94289	0.93679
	1 layer + GELU	0.93499	0.93613	0.93068
	1 layer + GELU + norm	0.94314	0.93775	0.94226
	1 layer + LeakyReLU	0.93618	0.93242	0.93536
	1 layer + ReLU	0.94000	0.93636	0.93218
	1 layer + ReLU + norm	0.93057	0.93819	0.94168
	2 layers + GELU	0.93225	0.91823	0.93331
	2 layers + GELU + norm	0.94279	0.94241	0.94281
	2 layers + LeakyReLU	0.93107	0.92393	0.93402
	2 layers + LeakyReLU + norm	0.93811	0.93258	0.93372
	2 layers + ReLU	0.93073	0.91607	0.93529
	2 layers + ReLU + norm	0.93299	0.93727	0.93620
Training process	lr 10^{-3} + batch 8	0.20114	0.21573	0.21573
	lr 10^{-7} + batch 8	0.30369	0.30316	0.29851
	lr 10^{-3} + batch 12	0.21573	0.20404	0.21573
	lr 10^{-5} + batch 12	0.93523	0.93443	0.93752
	lr 10^{-7} + batch 12	0.29594	0.30439	0.29565
	lr 10^{-3} + batch 16	0.21573	0.21573	0.21573
	lr 10^{-5} + batch 16	0.93049	0.93409	0.93441
	lr 10^{-7} + batch 16	0.29855	0.29698	0.29076

Table 8.3: F1-scores obtained by each configuration of the "*Model parameters*" experiments [6.1].

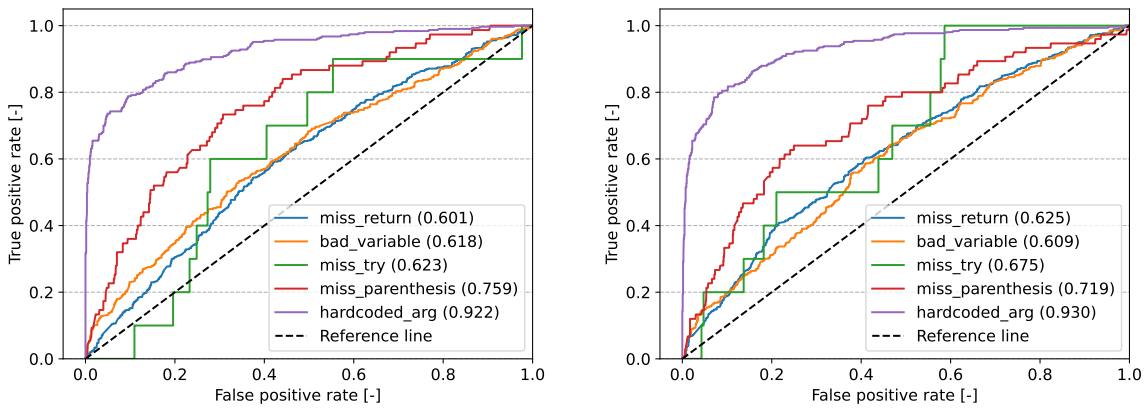
H Additional graphs for the "Model parameters" experiment

This appendix contains the graphs of the two other seeds we tried for each discussed configuration in the first experiment [6.1].



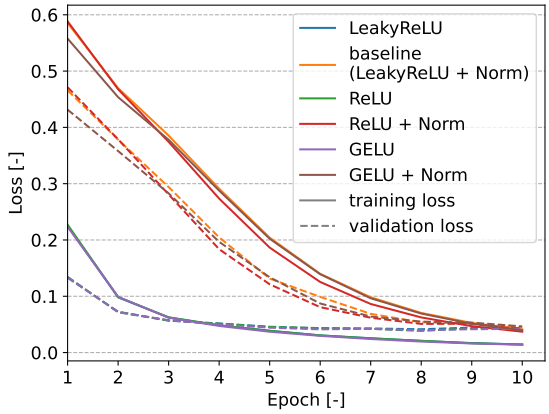
(a) Second seed (b) Third seed

Figure 8.18: Label-wise ROC curves for the *baseline*.

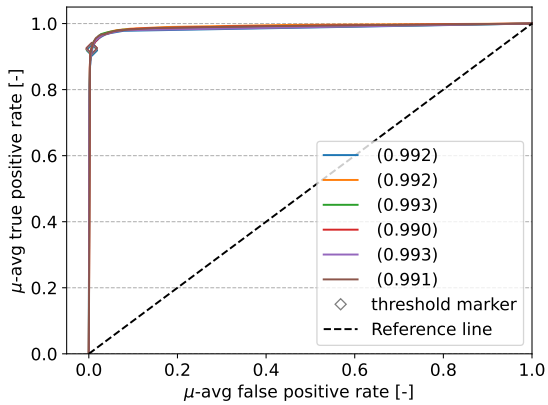


(a) Second seed (b) Third seed

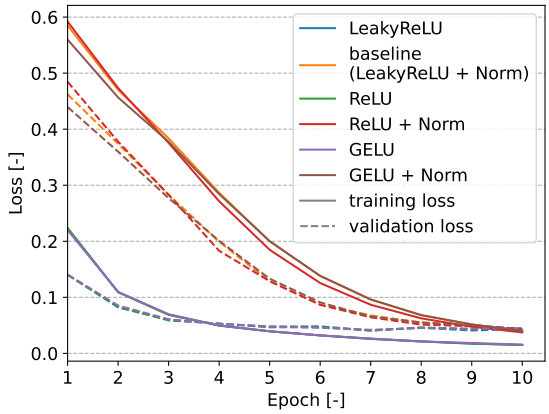
Figure 8.19: Label-wise ROC curves of the `[0, 512]` configuration.



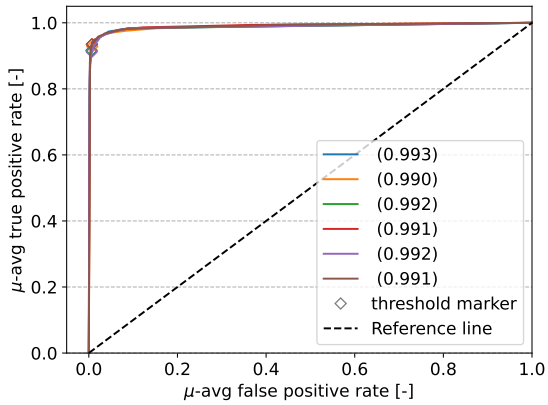
(a) Loss curves - Second seed



(b) ROC curves - Second seed (use the same color legend as 8.20a)

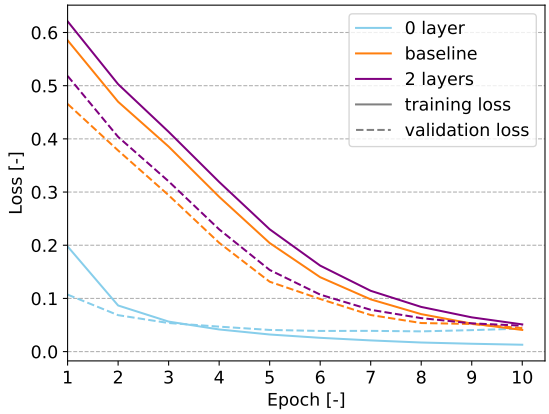


(c) Loss curves - Third seed

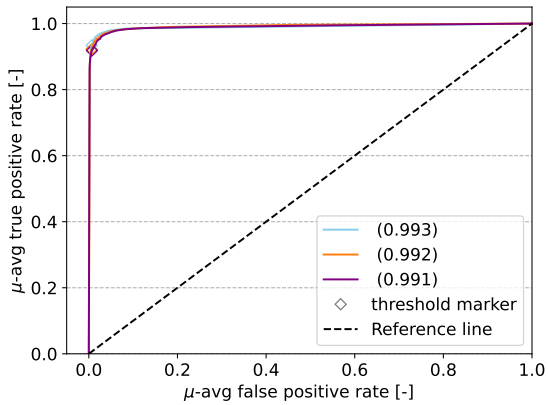


(d) ROC curves - Third seed (use the same color legend as 8.20c)

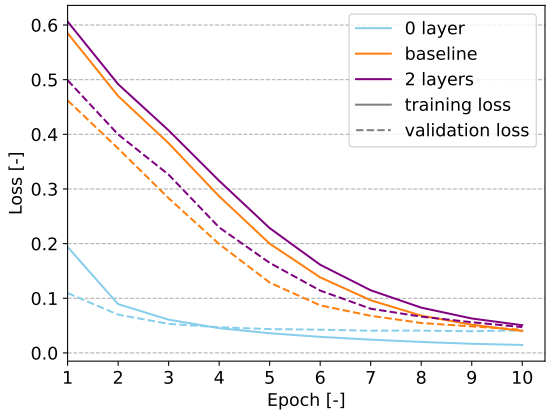
Figure 8.20: Results for the change in the batch-normalization and activation function in the classification head.



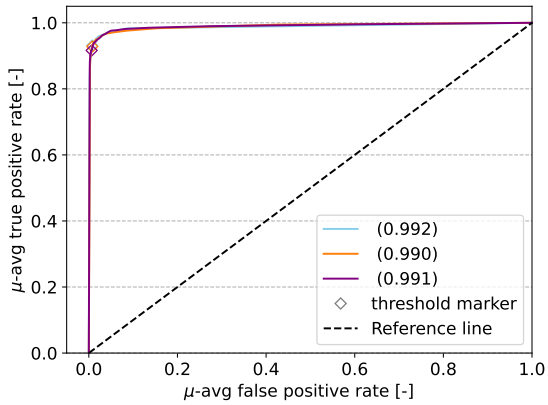
(a) Loss curves - Second seed



(b) ROC curves - Second seed (use the same color legend as 8.21a)

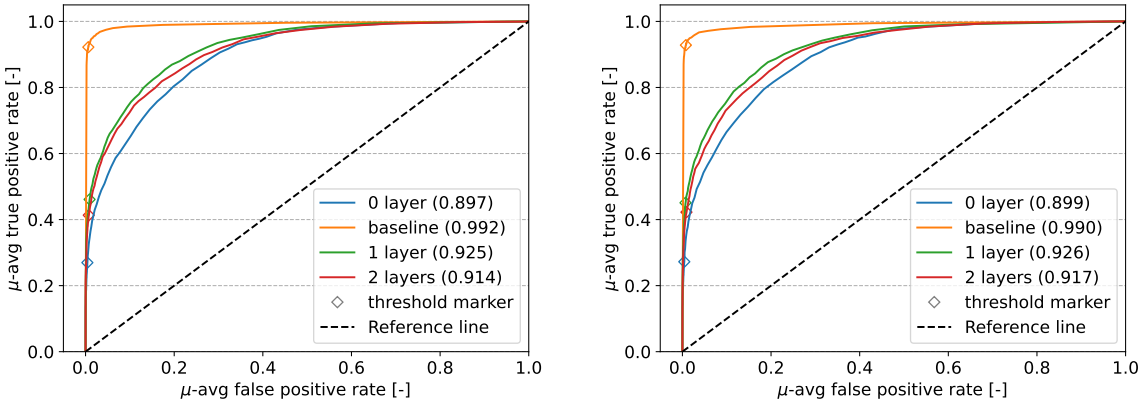


(c) Loss curves - Third seed



(d) ROC curves - Third seed (use the same color legend as 8.21c)

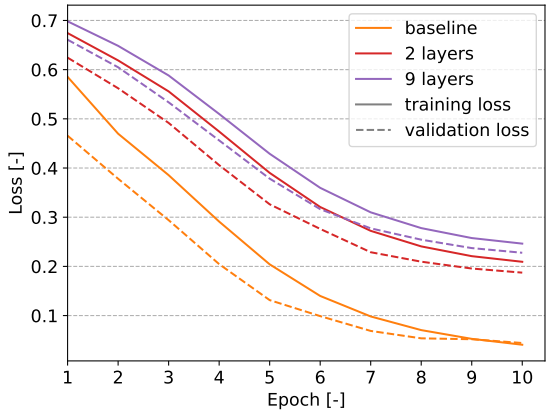
Figure 8.21: Results when changing the number of layers in the classification head.



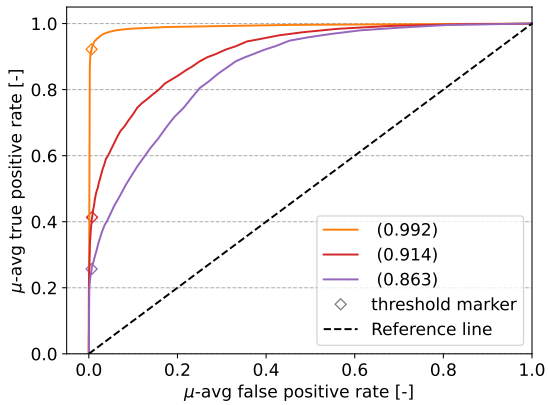
(a) Second seed

(b) Third seed

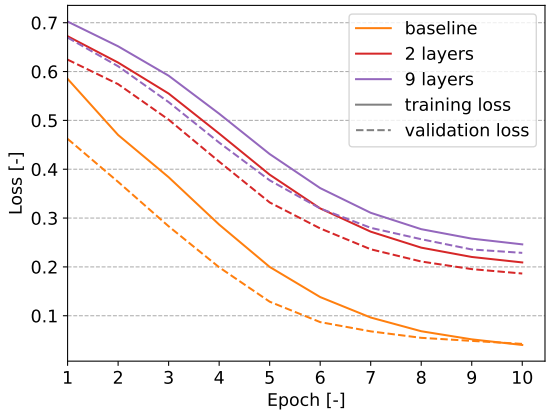
Figure 8.22: ROC curves when changing the number of layers without fine-tuning the embedder.



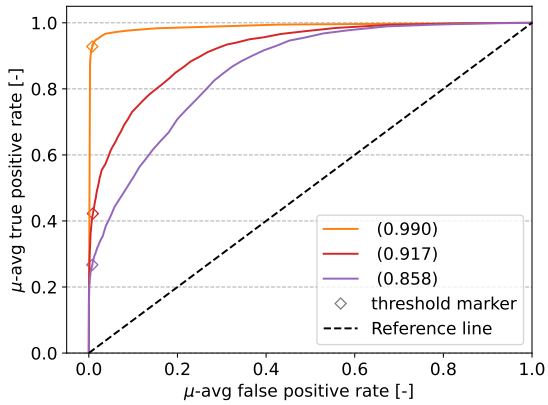
(a) Loss curves - Second seed



(b) ROC curves - Second seed (use the same color legend as 8.23a)

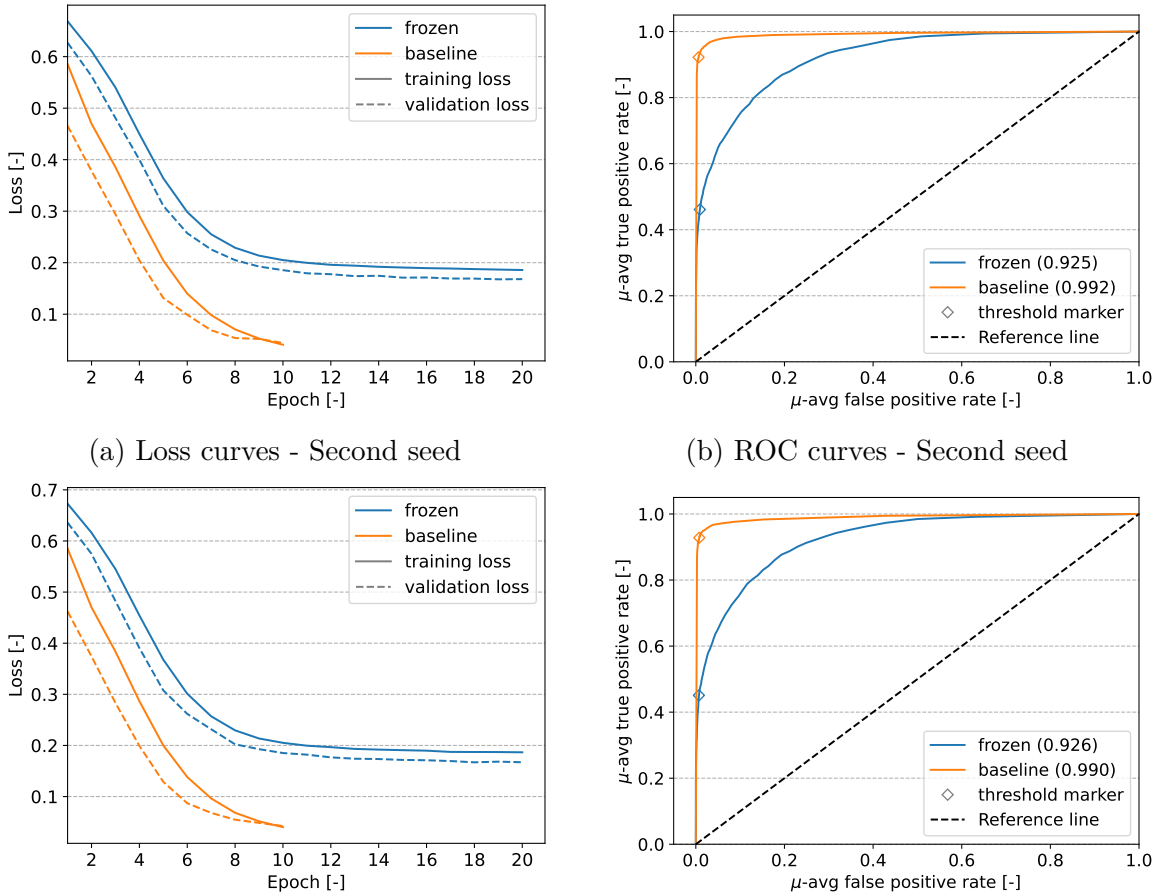


(c) Loss curves - Third seed



(d) ROC curves - Third seed (use the same color legend as 8.23c)

Figure 8.23: Comparison of the models without fine-tuning the embedder and using up to 9 of dense layers.



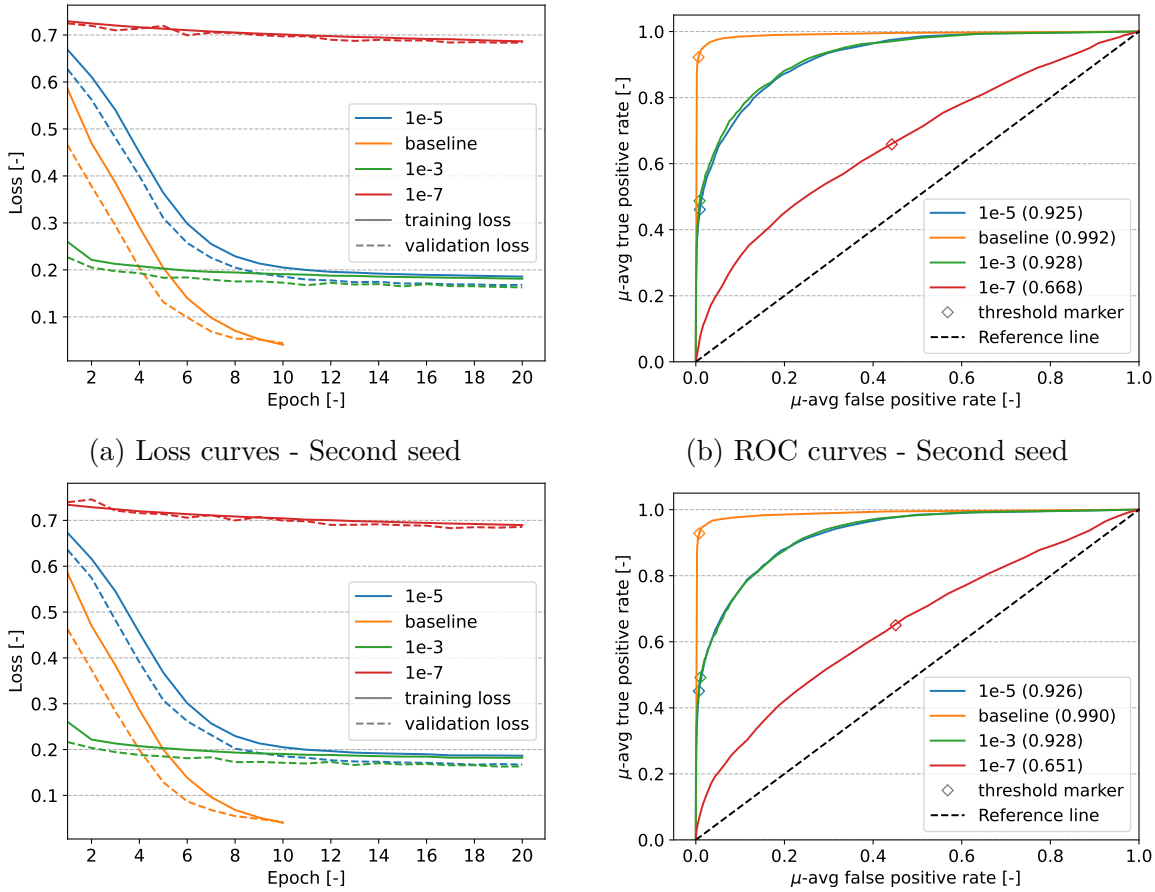
(a) Loss curves - Second seed

(b) ROC curves - Second seed

(c) Loss curves - Third seed

(d) ROC curves - Third seed

Figure 8.24: Comparison of the models without fine-tuning the embedder.



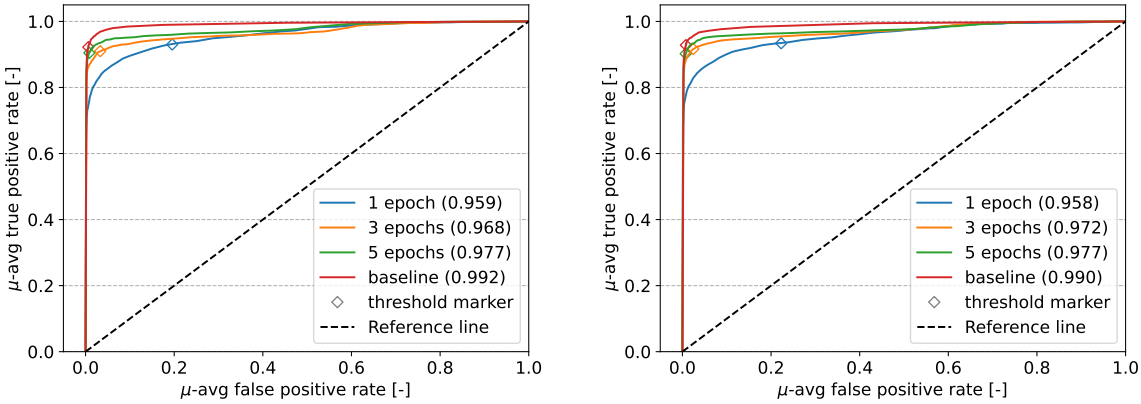
(a) Loss curves - Second seed

(b) ROC curves - Second seed

(c) Loss curves - Third seed

(d) ROC curves - Third seed

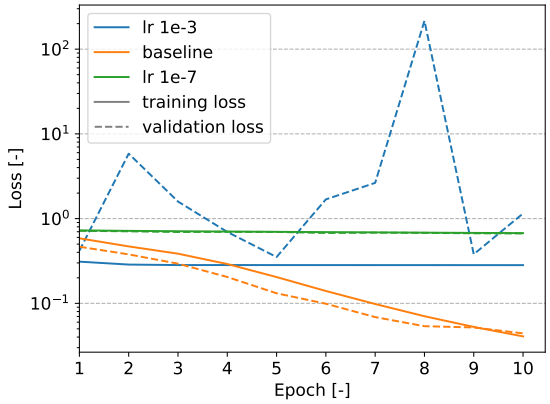
Figure 8.25: Results when varying the learning rate and freezing the embedder's weights.



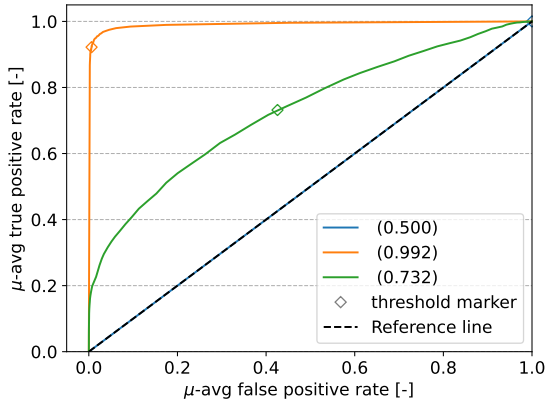
(a) Second seed

(b) Third seed

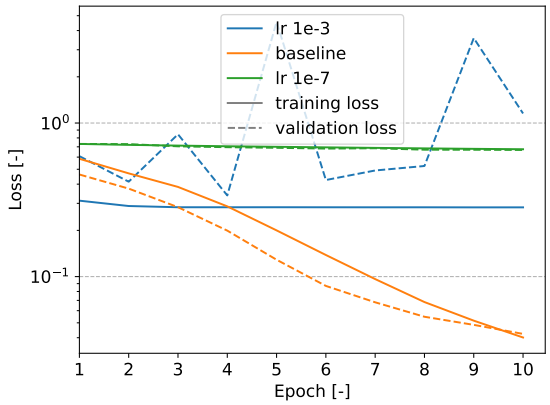
Figure 8.26: ROC curves when limiting the epochs while fine-tuning the embedder.



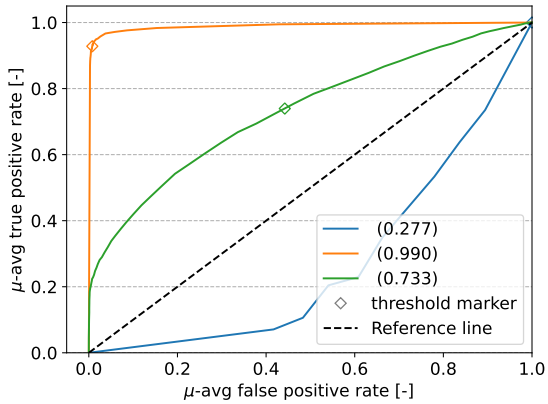
(a) Loss curves - Second seed



(b) ROC curves - Second seed (use the same color legend as 8.27a)



(c) Loss curves - Third seed



(d) ROC curves - Third seed (use the same color legend as 8.27c)

Figure 8.27: Results when changing the learning rate.

I Evaluation dataset statistics

Class	Occurrence number	Label proportion	Instance proportion
`miss_return`	58	19.205%	36.025%
`bad_variable`	58	19.205%	36.025%
`print_return`	40	15.232%	24.845%
`bad_loop`	25	8.278%	15.528%
`failed`	24	7.947%	14.907%
`miss_try`	18	5.960%	11.180%
`bad_file`	17	5.629%	10.559%
`bad_range`	16	5.298%	9.938%
`bad_list`	12	3.974%	7.453%
`bad_assign`	11	3.642%	6.832%
`miss_loop`	11	3.642%	6.832%
`bad_division`	5	1.656%	3.106%
`hardcoded_arg`	4	1.325%	2.484%
`miss_parenthesis`	2	0.662%	1.242%
`overwrite_not_increment`	1	0.331%	0.621%
`correct`	0	0.000%	0.000%
TOTAL:	302		

Table 8.4: Statistics for the evaluation dataset of the second experiment [6.2].

J Evaluation dataset *baseline* results

The following pages present the labels predicted by each run of the *baseline* configuration. The name `o-n-i` is the abbreviation of `overwrite_not_increment` and `miss-par` of `miss_parenthesis`. Inst. means "Instance" and corresponds to the instance number inside our evaluation dataset. The "Run" columns correspond to the predicted outputs of each run.

Inst.	Ground Truth	Run 1	Run 2	Run 3
1	- bad_variable	- failed	- bad_division - miss_loop	- failed
2	- bad_range		- failed	
3	- bad_variable	- bad_variable - bad_loop - miss_loop	- bad_loop - miss_loop	- bad_variable - bad_loop - miss_loop
4	- failed		- bad_variable	
5	- bad_division - miss-par	- bad_division	- failed	- bad_division
6	- o-n-i			- failed
7	- failed	- bad_variable	- bad_variable	- bad_variable
8	- bad_loop - bad_assign	- bad_loop - bad_assign	- bad_loop - bad_assign	- bad_loop - bad_assign
9	- print_return - miss_return - hardcoded_arg	- miss_return - print_return - hardcoded_arg	- miss_return - print_return - hardcoded_arg	- miss_return - print_return - hardcoded_arg
10	- failed	- bad_division - bad_loop	- bad_division - miss_loop	- bad_division
11	- failed - miss_loop		- correct	- failed
12	- bad_variable - bad_loop - bad_assign	- bad_loop - bad_assign	- bad_loop - bad_assign	- bad_loop - bad_assign
13	- miss_return	- miss_return	- miss_return	- miss_return

				- miss_loop
14	- bad_list	- bad_list - miss-par	- bad_list	- bad_list
15	- bad_variable - miss_return	- bad_division	- bad_division - bad_variable	- miss_return - bad_division - bad_variable
16	- miss_return - print_return - bad_range	- miss_return - bad_range	- miss_return - print_return - bad_range	- miss_return - print_return - bad_range
17	- bad_list - bad_assign - bad_loop	- bad_loop - bad_assign	- bad_loop - bad_assign	- bad_loop - bad_assign
18	- failed	- failed	- failed	- failed
19	- miss_try - bad_loop	- bad_loop	- bad_loop	- failed
20	- bad_variable - failed	- bad_variable	- failed	- failed
21	- print_return - miss_return	- miss_return - print_return	- miss_return - print_return	- miss_return - print_return
22	- bad_file - miss_try	- bad_file	- failed	- failed
23	- miss_try - bad_variable	- failed	- failed	- failed
24	- bad_list - miss_try - bad_variable	- bad_variable - bad_list	- bad_variable - bad_list	- bad_list
25	- miss_return - miss_try	- miss_return	- miss_return	- miss_return - bad_file
26	- failed - miss_try	- failed	- correct	- failed
27	- miss_return	- miss_return	- miss_return	- miss_return

	- print_return - bad_variable - miss_try	- miss_loop	- print_return - miss_loop	- print_return - miss_loop
28	- failed			- bad_division
29	- miss_return - print_return	- miss_return - print_return - miss_loop	- miss_return - print_return	- miss_return - print_return
30	- bad_variable		- miss_loop	
31	- print_return - bad_assign - bad_division	- miss_return - print_return - bad_assign	- miss_return - print_return - bad_assign	- miss_return - print_return - bad_assign
32	- bad_loop	- bad_loop	- bad_loop	- bad_loop
33	- hardcoded_arg - bad_variable - bad_loop - miss_return	- miss_return - bad_loop	- miss_return - bad_loop	- miss_return - bad_loop - miss_loop
34	- bad_loop	- bad_loop	- bad_loop	- bad_loop
35	- bad_list	- bad_list	- bad_list	- bad_list
36	- bad_variable	- bad_variable	- bad_variable	- bad_variable
37	- bad_list	- bad_list	- bad_list	- bad_list
38	- failed	- bad_list	- bad_list	- bad_list
39	- miss_loop	- miss_loop	- bad_loop - miss_loop	- miss_loop
40	- miss_return - bad_range	- miss_return - bad_range	- miss_return - bad_range	- miss_return - bad_range
41	- bad_variable - print_return - miss_return - bad_loop	- miss_return - bad_loop	- miss_return - print_return - bad_loop	- miss_return - print_return
42	- miss_return	- miss_return	- miss_return	- miss_return
43	- hardcoded_arg	- hardcoded_arg	- o-n-i	- miss_loop

	- miss_loop	- miss_loop	- miss_loop	
44	- bad_assign	- bad_assign - o-n-i	- bad_assign	- bad_assign
45	- miss_try	- miss_try	- miss_try	- miss_try
46	- bad_range - bad_list	- bad_range - o-n-i	- bad_range - o-n-i	- bad_range - o-n-i
47	- miss_return	- miss_return	- miss_return	- miss_return
48	- bad_variable	- bad_variable	- bad_variable	- bad_variable
49	- bad_assign - print_return - miss_return	- miss_return - print_return - bad_assign	- miss_return - print_return - bad_assign	- miss_return - print_return - bad_assign
50	- bad_file	- bad_file	- failed	- failed
51	- miss_return - print_return	- miss_return - print_return	- miss_return - print_return	- miss_return - print_return
52	- bad_variable - miss_return - print_return	- miss_return	- miss_return - print_return - bad_variable	- miss_return - print_return
53	- bad_variable	- bad_variable	- bad_variable	- bad_variable - failed
54	- bad_loop	- bad_loop	- bad_loop	- bad_loop
55	- bad_variable - bad_loop	- miss_return - bad_variable	- miss_return - bad_variable	- miss_return - bad_variable
56	- bad_variable	- miss_return - bad_variable	- miss_return - bad_variable	- miss_return - miss_loop
57	- bad_assign - bad_variable	- bad_assign	- bad_assign - miss_loop	- bad_variable - bad_assign
58	- bad_variable - miss_return - print_return - bad_loop	- miss_return - print_return - bad_loop	- miss_return - print_return - bad_loop	- miss_return - print_return - bad_loop
59	- bad_variable	- miss_return	- miss_return	- miss_return

	- miss_return - print_return	- print_return	- print_return	- print_return
60	- failed			- failed
61	- bad_variable	- bad_variable	- bad_variable	- bad_variable - o-n-i
62	- failed - bad_range	- correct	- correct	- failed
63	- failed	- correct	- correct	- failed
64	- miss_return	- miss_return	- miss_return	- miss_return
65	- miss_loop	- miss_loop	- miss_loop	- miss_loop
66	- miss_return - print_return - bad_range	- miss_return - print_return	- miss_return - print_return - bad_range	- miss_return - print_return - bad_range
67	- failed	- failed	- correct	- o-n-i
68	- bad_variable - failed	- failed	- failed	- failed
69	- bad_variable	- failed	- failed	- o-n-i - failed
70	- print_return - miss_return	- miss_return - print_return	- miss_return - print_return	- miss_return - print_return
71	- bad_loop - miss_try - bad_variable	- bad_variable - bad_loop	- bad_variable - bad_loop	- bad_variable - bad_loop
72	- bad_variable		- failed	- bad_variable
73	- bad_file - bad_loop - miss_try	- bad_loop - bad_file	- bad_loop	- bad_loop - bad_file
74	- miss_try - bad_variable - bad_file	- bad_variable - miss_try	- bad_variable - miss_try	- miss_try
75	- miss_try	- miss_return	- miss_return	- miss_return

	- miss_return - print_return - bad_variable	- bad_variable	- print_return - bad_variable	- print_return - bad_variable
76	- bad_variable - miss_return - print_return - bad_loop - bad_file	- miss_return - print_return - bad_loop	- miss_return - print_return - bad_loop	- miss_return - print_return - bad_loop
77	- print_return - miss_return	- miss_return - print_return	- miss_return - print_return	- miss_return - print_return
78	- bad_file - bad_variable - miss_return - print_return	- miss_return - print_return	- miss_return - print_return	- miss_return - print_return - o-n-i
79	- bad_loop	- bad_loop	- bad_loop	- bad_loop
80	- bad_variable - bad_loop	- bad_variable - bad_loop	- bad_loop	- bad_variable - bad_loop
81	- bad_list	- bad_list	- bad_list	- bad_list - miss_loop
82	- bad_assign - bad_division	- bad_division - bad_assign	- bad_assign	- bad_division - bad_assign
83	- bad_range	- bad_range	- bad_range	- bad_range
84	- bad_variable - miss_try	- failed		- failed
85	- bad_variable - miss_return - print_return - bad_file	- miss_return - print_return	- miss_return - print_return	- miss_return - print_return
86	- miss_loop	- failed	- correct	- failed
87	- failed			- failed
88	- miss_return	- miss_return	- miss_return	- miss_return

	- print_return		- bad_range	- print_return - bad_range
89	- bad_variable - bad_range	- correct	- correct	- failed
90	- miss_return - print_return - bad_file - bad_variable - miss_try	- miss_return - bad_file	- miss_return - print_return	- miss_return - print_return - bad_file
91	- bad_variable - miss_try	- failed	- correct	- failed
92	- bad_file - miss_try	- bad_file - failed	- failed	- failed
93	- miss_return - print_return	- miss_return - print_return	- miss_return - print_return	- miss_return - print_return
94	- miss_return	- miss_return	- miss_return	- miss_return
95	- failed		- failed	- failed
96	- miss_return - bad_variable	- miss_return - bad_variable	- miss_return	- miss_return
97	- miss_return - print_return	- miss_return - print_return	- miss_return - print_return	- miss_return - print_return
98	- bad_file - miss_try		- o-n-i	
99	- failed	- failed	- failed	- failed
100	- bad_file - bad_variable	- failed	- failed	- failed
101	- bad_range	- bad_range - bad_list	- bad_range	- bad_range
102	- failed	- bad_list	- bad_list	- bad_list
103	- bad_variable - miss-par	- bad_variable	- bad_variable	- bad_variable

	- bad_range			
104	- bad_range	- bad_variable	- bad_variable	- miss_return - bad_variable
105	- bad_range - bad_list	- bad_range	- bad_range	- bad_range
106	- failed	- correct	- correct	
107	- bad_range	- bad_range	- bad_range	- bad_range
108	- bad_loop		- bad_variable	- failed
109	- failed	- failed	- failed	- failed
110	- miss_return	- miss_return	- miss_return	- miss_return
111	- miss_return - print_return - bad_file	- miss_return	- miss_return - print_return	- miss_return - print_return
112	- failed	- correct	- correct	- failed
113	- bad_file			- bad_file
114	- miss_return - print_return - bad_variable	- miss_return - print_return - bad_variable	- miss_return - print_return - bad_variable	- miss_return - print_return
115	- miss_loop - miss_return - print_return	- miss_return - miss_loop	- miss_return - print_return - miss_loop	- miss_return - print_return - miss_loop
116	- miss_return - print_return	- miss_return - print_return - bad_variable	- miss_return - print_return - bad_variable	- miss_return - print_return
117	- bad_loop - miss_return	- miss_return - bad_loop	- bad_loop	- bad_loop
118	- bad_loop - miss_return	- miss_return - miss_loop	- miss_return	- miss_return - print_return
119	- miss_return - print_return - bad_loop	- miss_return - bad_loop	- miss_return - print_return - bad_variable	- miss_return - print_return - bad_variable

	- bad_variable		- bad_loop	- bad_loop
120	- miss_return - print_return	- miss_return	- miss_return - print_return	- miss_return - print_return
121	- bad_list	- bad_list	- bad_list	- bad_list
122	- bad_loop	- bad_loop - bad_range	- bad_loop	- bad_loop - bad_range
123	- miss_return - bad_list	- miss_return	- miss_return - bad_list	- miss_return
124	- print_return - miss_return	- miss_return	- miss_return - print_return	- miss_return - print_return
125	- bad_variable		- correct	- failed - correct
126	- miss_return - bad_variable - print_return	- miss_return - bad_variable	- miss_return - print_return - bad_variable	- miss_return - print_return - bad_variable
127	- miss_return - print_return - bad_loop	- miss_return - print_return - bad_loop	- miss_return - print_return - bad_loop	- miss_return - print_return
128	- bad_variable	- o-n-i	- o-n-i	- o-n-i - failed
129	- bad_variable	- bad_variable	- correct	- failed
130	- bad_range - bad_variable	- bad_variable - bad_range	- bad_range	- bad_range
131	- bad_variable		- miss_return	- miss_return
132	- miss_return - bad_variable	- miss_return - bad_variable	- miss_return - bad_variable	- miss_return - bad_variable
133	- bad_variable - miss_loop - miss_return	- miss_return - miss_loop	- miss_return - miss_loop	- miss_return - miss_loop
134	- print_return - miss_return	- miss_return	- miss_return	- miss_return - print_return

	- bad_variable			
135	- miss_loop - bad_variable	- miss_return - bad_variable	- miss_return - bad_variable	- miss_return - bad_variable
136	- failed			- failed
137	- bad_division	- bad_division	- bad_division	- bad_division
138	- bad_division	- correct	- bad_division	
139	- miss_return - bad_range	- miss_return - bad_range	- miss_return - bad_range	- miss_return - bad_range
140	- bad_range		- bad_range	- bad_range
141	- bad_assign - hardcoded_arg - miss_loop - miss_return - print_return	- miss_return - print_return - bad_assign - miss_loop	- miss_return - print_return - bad_assign - miss_loop	- miss_return - print_return - bad_assign - miss_loop
142	- bad_list	- bad_list	- bad_list	- bad_list
143	- bad_assign - bad_loop	- bad_loop - bad_assign	- bad_loop - bad_assign - miss_loop	- bad_loop - bad_assign - miss_loop
144	- bad_variable - bad_list	- bad_variable - bad_list	- bad_variable - bad_list	- bad_variable - bad_list
145	- bad_variable - bad_file	- bad_variable		- bad_variable
146	- bad_file	- bad_file	- failed	- failed
147	- bad_file - miss_return - print_return - bad_variable	- miss_return - print_return - bad_variable	- miss_return - print_return - bad_variable	- miss_return - print_return - bad_variable
148	- bad_file - failed	- failed	- failed	- failed
149	- miss_loop	- miss_loop	- bad_loop - miss_loop	- miss_loop

150	- miss_try	- miss_try	- miss_try - correct	- miss_try
151	- bad_variable - miss_return - print_return	- miss_return - print_return	- miss_return - print_return	- miss_return - print_return
152	- miss_return - print_return	- miss_return	- miss_return - print_return	- miss_return - print_return
153	- miss_return - print_return - bad_variable - bad_assign	- miss_return - print_return - bad_assign	- miss_return - print_return - bad_assign	- miss_return - print_return - bad_range - bad_assign
154	- failed	- bad_range	- bad_range	- bad_range
155	- bad_variable	- bad_variable	- bad_variable - bad_range	- bad_variable
156	- miss_return	- miss_return	- miss_return	- miss_return
157	- miss_return - print_return - bad_variable	- miss_return - bad_variable	- miss_return - print_return - bad_variable	- miss_return - print_return - bad_variable
158	- miss_return - bad_variable - miss_loop	- miss_return - miss_loop	- miss_return - miss_loop	- miss_return - miss_loop
159	- miss_return - print_return	- miss_return - print_return	- miss_return - print_return	- miss_return - print_return
160	- bad_loop	- bad_variable - bad_loop	- bad_loop	- bad_loop
161	- bad_loop	- bad_loop		- bad_loop

K Label-wise results of *baseline* on the evaluation dataset

This first table presents the average scores the *baseline* obtained over its runs.

Label	Precision	Recall	F1-score
`bad_assign`	1.00000	1.00000	1.00000
`miss_return`	0.91907	0.97701	0.94710
`print_return`	0.99187	0.85000	0.90400
`bad_loop`	0.91389	0.82667	0.86652
`bad_list`	0.78182	0.69444	0.73518
`bad_range`	0.77908	0.64583	0.70323
`miss_loop`	0.57778	0.72727	0.64214
`bad_variable`	0.86605	0.40230	0.54838
`bad_division`	0.50000	0.53333	0.51515
`hardcoded_arg`	1.00000	0.33333	0.48889
`failed`	0.43987	0.40278	0.40589
`miss_try`	1.00000	0.16667	0.28571
`bad_file`	0.58333	0.17647	0.26915
`miss_parenthesis`	0.00000	0.00000	0.00000
`overwrite_not_increment`	0.00000	0.00000	0.00000
`correct`	0.00000	0.00000	0.00000

Table 8.6: Average scores obtained by the *baseline* on the evaluation dataset [6.2].

This second table presents the standard deviation of each metric.

Label	Precision	Recall	F1-score
bad_assign	0.00000	0.00000	0.00000
miss_return	0.01486	0.00995	0.00944
print_return	0.01408	0.21794	0.13354
bad_loop	0.03758	0.06110	0.02796
bad_list	0.04810	0.04811	0.04401
bad_range	0.04281	0.07217	0.03380
miss_loop	0.07698	0.00000	0.04634
bad_variable	0.05093	0.03982	0.03803
bad_division	0.10000	0.11547	0.10339
hardcoded_arg	0.00000	0.14434	0.15396
failed	0.02747	0.19245	0.09683
miss_try	0.00000	0.00000	0.00000
bad_file	0.52042	0.17647	0.26126
miss_parenthesis	0.00000	0.00000	0.00000
overwrite_not_increment	0.00000	0.00000	0.00000
correct	0.00000	0.00000	0.00000

Table 8.7: Standard deviation of scores obtained by the *baseline* on the evaluation dataset [6.2].

L Other datasets statistics

This appendix presents the number of instances associated with each label in the other datasets explored in **RQ2.3** [6.2]. The name `o-n-i` is the abbreviation of `overwrite_not_increment` and `miss-par` of `miss_parenthesis`. The percentages represent the proportion of instances in which the label appears.

Labels	Train	Validation	Test	Total
- `miss_return`	8550 (35.647%)	921 (34.913%)	1059 (35.288%)	10530 (35.546%)
- `hardcoded_arg`	5104 (21.280%)	545 (20.660%)	635 (21.160%)	6284 (21.213%)
- `bad_loop`	5065 (21.117%)	556 (21.077%)	662 (22.059%)	6283 (21.209%)
- `bad_variable`	4936 (20.580%)	569 (21.569%)	660 (21.993%)	6165 (20.811%)
- `bad_assign`	4728 (19.712%)	529 (20.053%)	561 (18.694%)	5818 (19.639%)
- `print_return`	4284 (17.861%)	461 (17.475%)	512 (17.061%)	5257 (17.746%)
- `miss_loop`	3977 (16.581%)	433 (16.414%)	494 (16.461%)	4904 (16.554%)
- `bad_range`	3575 (14.905%)	419 (15.883%)	435 (14.495%)	4429 (14.951%)
- `bad_list`	1829 (7.626%)	195 (7.392%)	179 (5.965%)	2203 (7.437%)
- `miss-par`	1007 (4.198%)	113 (4.284%)	144 (4.798%)	1264 (4.267%)
- `o-n-i`	992 (4.136%)	109 (4.132%)	126 (4.199%)	1227 (4.142%)
- `correct`	680 (2.835%)	83 (3.146%)	92 (3.066%)	855 (2.886%)
- `bad_division`	512 (2.135%)	39 (1.478%)	75 (2.499%)	626 (2.113%)
- `bad_file`	415 (1.730%)	46 (1.744%)	37 (1.233%)	498 (1.681%)
- `failed`	293 (1.222%)	34 (1.289%)	37 (1.233%)	364 (1.229%)
- `miss_try`	232 (0.967%)	25 (0.948%)	33 (1.100%)	290 (0.979%)

Table 8.8: Statistics for the `more_mutants` dataset. This dataset has 23985 train instances, 2638 for the validation set and 3001 for the test set.

Labels	Train	Validation	Test	Total
- `miss_return`	28193 (34.703%)	3084 (34.466%)	3495 (34.662%)	34772 (34.678%)
- `hardcoded_arg`	16704 (20.561%)	1836 (20.519%)	2089 (20.718%)	20629 (20.573%)
- `bad_loop`	16606 (20.440%)	1838 (20.541%)	2079 (20.619%)	20523 (20.467%)
- `bad_variable`	16613 (20.449%)	1831 (20.463%)	2073 (20.559%)	20517 (20.461%)
- `bad_assign`	15880 (19.547%)	1773 (19.814%)	1961 (19.449%)	19614 (19.561%)
- `miss_loop`	14325 (17.633%)	1618 (18.082%)	1763 (17.485%)	17706 (17.658%)
- `print_return`	13960 (17.183%)	1530 (17.099%)	1724 (17.098%)	17214 (17.167%)
- `bad_range`	12822 (15.783%)	1506 (16.831%)	1574 (15.610%)	15902 (15.859%)
- `bad_list`	7474 (9.200%)	774 (8.650%)	823 (8.162%)	9071 (9.046%)
- `o-n-i`	4208 (5.180%)	439 (4.906%)	514 (5.098%)	5161 (5.147%)
- `miss-par`	3671 (4.519%)	434 (4.850%)	523 (5.187%)	4628 (4.615%)
- `correct`	2018 (2.484%)	223 (2.492%)	249 (2.470%)	2490 (2.483%)
- `bad_division`	1853 (2.281%)	134 (1.498%)	294 (2.916%)	2281 (2.275%)
- `bad_file`	1611 (1.983%)	162 (1.810%)	186 (1.845%)	1959 (1.954%)
- `miss_try`	1192 (1.467%)	114 (1.274%)	141 (1.398%)	1447 (1.443%)
- `failed`	293 (0.361%)	34 (0.380%)	37 (0.367%)	364 (0.363%)

Table 8.9: Statistics for the `all_mutants` dataset. This dataset has 81241 train instances, 8948 for the validation set and 10083 for the test set.

Labels	Train	Validation	Test	Total
- miss_return	8474 (37.406%)	960 (38.035%)	1105 (39.478%)	10539 (37.670%)
- bad_loop	4980 (21.983%)	555 (21.989%)	598 (21.365%)	6133 (21.922%)
- bad_variable	4563 (20.142%)	527 (20.880%)	541 (19.328%)	5631 (20.127%)
- hardcoded_arg	4529 (19.992%)	491 (19.453%)	523 (18.685%)	5543 (19.813%)
- print_return	4405 (19.445%)	457 (18.106%)	560 (20.007%)	5422 (19.380%)
- bad_assign	4245 (18.738%)	503 (19.929%)	529 (18.900%)	5277 (18.862%)
- miss_loop	3589 (15.843%)	360 (14.263%)	427 (15.255%)	4376 (15.641%)
- bad_range	3097 (13.671%)	334 (13.233%)	381 (13.612%)	3812 (13.625%)
- bad_list	1721 (7.597%)	177 (7.013%)	194 (6.931%)	2092 (7.478%)
- miss-par	934 (4.123%)	107 (4.239%)	118 (4.216%)	1159 (4.143%)
- o-n-i	870 (3.840%)	105 (4.160%)	98 (3.501%)	1073 (3.835%)
- correct	519 (2.291%)	63 (2.496%)	63 (2.251%)	645 (2.305%)
- bad_file	514 (2.269%)	56 (2.219%)	54 (1.929%)	624 (2.230%)
- bad_division	478 (2.110%)	49 (1.941%)	72 (2.572%)	599 (2.141%)
- failed	440 (1.942%)	45 (1.783%)	56 (2.001%)	541 (1.934%)
- miss_try	244 (1.077%)	31 (1.228%)	23 (0.822%)	298 (1.065%)

Table 8.10: Statistics for the `more_codes` dataset. This dataset has 22654 train instances, 2524 for the validation set and 2799 for the test set.

Labels	Train	Validation	Test	Total
- miss_return	29644 (38.429%)	3339 (38.780%)	3753 (39.406%)	36736 (38.558%)
- bad_loop	16887 (21.891%)	1841 (21.382%)	2110 (22.155%)	20838 (21.872%)
- hardcoded_arg	16045 (20.800%)	1772 (20.581%)	1921 (20.170%)	19738 (20.717%)
- bad_variable	15850 (20.547%)	1826 (21.208%)	1929 (20.254%)	19605 (20.577%)
- print_return	15462 (20.044%)	1728 (20.070%)	1916 (20.118%)	19106 (20.054%)
- bad_assign	15145 (19.633%)	1717 (19.942%)	1835 (19.267%)	18697 (19.624%)
- miss_loop	12394 (16.067%)	1307 (15.180%)	1536 (16.128%)	15237 (15.993%)
- bad_range	10946 (14.190%)	1232 (14.309%)	1368 (14.364%)	13546 (14.218%)
- bad_list	6041 (7.831%)	638 (7.410%)	688 (7.224%)	7367 (7.732%)
- miss-par	3255 (4.220%)	378 (4.390%)	447 (4.693%)	4080 (4.282%)
- o-n-i	3028 (3.925%)	333 (3.868%)	341 (3.580%)	3702 (3.886%)
- bad_division	1674 (2.170%)	200 (2.323%)	230 (2.415%)	2104 (2.208%)
- bad_file	1387 (1.798%)	173 (2.009%)	164 (1.722%)	1724 (1.810%)
- correct	1333 (1.728%)	159 (1.847%)	163 (1.711%)	1655 (1.737%)
- miss_try	782 (1.014%)	105 (1.220%)	108 (1.134%)	995 (1.044%)
- failed	440 (0.570%)	45 (0.523%)	56 (0.588%)	541 (0.568%)

Table 8.11: Statistics for the `all_codes` dataset. This dataset has 77140 train instances, 8610 for the validation set and 9524 for the test set.

M Label-wise results for model limited to 5 epochs on the evaluation dataset

This first table presents the average scores the model limited on 5 epochs obtained over its runs.

Label	Precision	Recall	F1-score
bad_assign	1.00000	1.00000	1.00000
print_return	0.97541	0.99167	0.98344
miss_return	0.93018	0.99425	0.96111
bad_loop	0.95514	0.82667	0.88587
bad_range	0.87302	0.66667	0.75414
bad_list	0.69365	0.69444	0.69075
miss_loop	0.68214	0.63636	0.64942
bad_variable	0.81403	0.46552	0.58663
hardcoded_arg	1.00000	0.41667	0.57778
bad_file	0.91414	0.43137	0.57747
bad_division	0.53333	0.26667	0.31111
failed	0.32484	0.31944	0.30917
miss_try	0.00000	0.00000	0.00000
miss_parenthesis	0.00000	0.00000	0.00000
overwrite_not_increment	0.00000	0.00000	0.00000
correct	0.00000	0.00000	0.00000

Table 8.12: Average scores obtained by the *baseline* limited to 5 epochs on the evaluation dataset [6.2].

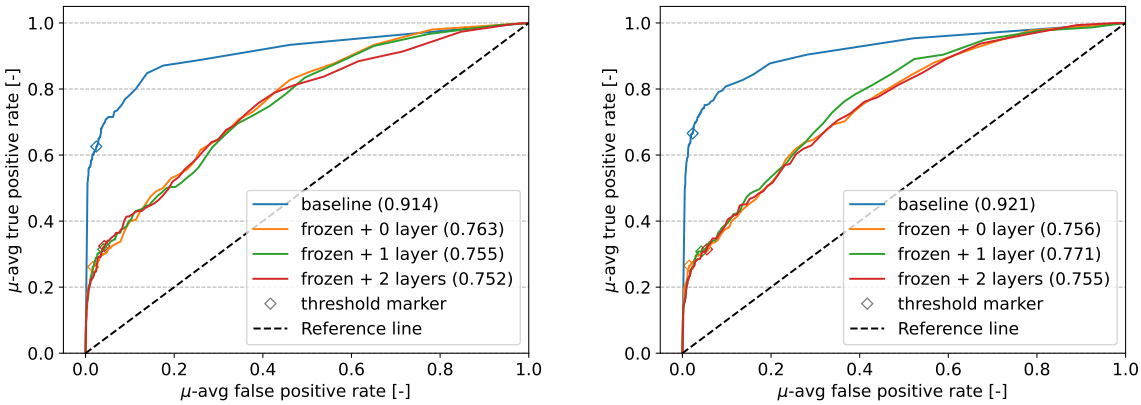
This second table presents the standard deviation of each metric.

Label	Precision	Recall	F1-score
bad_assign	0.00000	0.00000	0.00000
print_return	0.00035	0.01443	0.00731
miss_return	0.00828	0.00995	0.00483
bad_loop	0.04354	0.02309	0.02369
bad_range	0.11252	0.03608	0.05340
bad_list	0.02444	0.12729	0.06963
miss_loop	0.16763	0.09091	0.08311
bad_variable	0.08190	0.07901	0.05457
hardcoded_arg	0.00000	0.14434	0.15396
bad_file	0.08345	0.14804	0.13984
bad_division	0.50332	0.30551	0.30062
failed	0.28507	0.31273	0.27304
miss_try	0.00000	0.00000	0.00000
miss_parenthesis	0.00000	0.00000	0.00000
overwrite_not_increment	0.00000	0.00000	0.00000
correct	0.00000	0.00000	0.00000

Table 8.13: Standard deviation of each score obtained by the *baseline* limited to 5 epochs on the evaluation dataset [6.2].

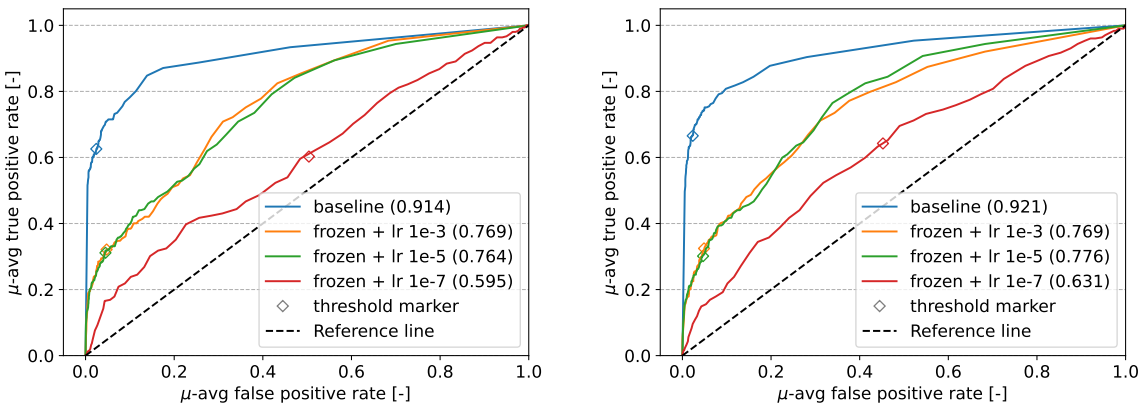
N Additional graphs for the "Model validation" experiment

This appendix contains the graphs of the two other seeds we tried for each discussed configuration in the second experiment [6.2].



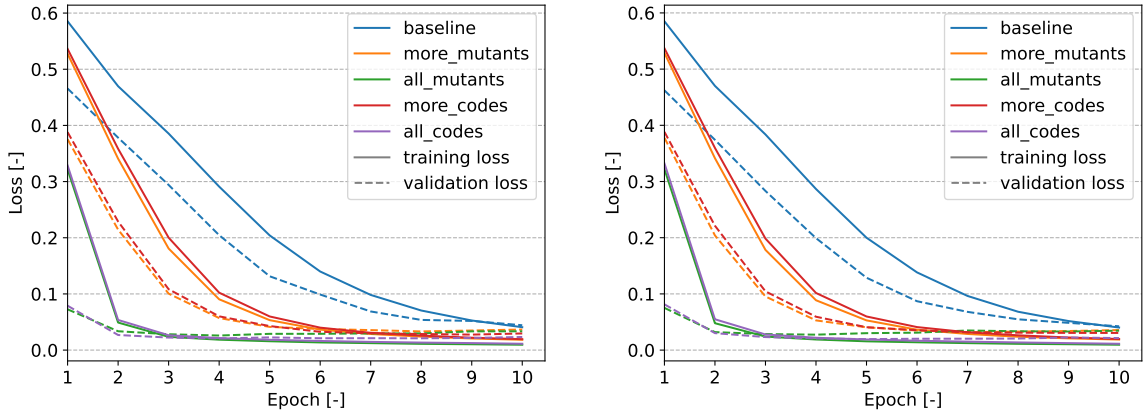
(a) Second seed (b) Third seed

Figure 8.28: Results on the evaluation dataset for the models without embedder fine-tuning when varying the number of layers.



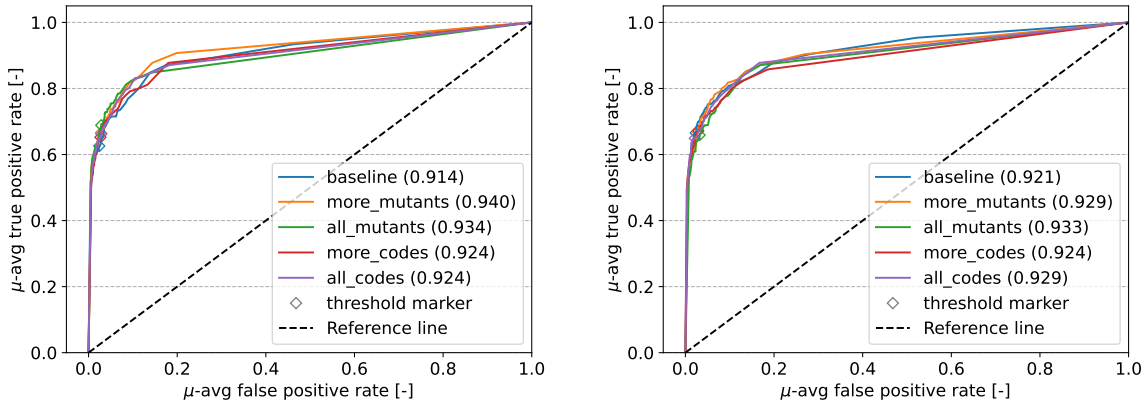
(a) Second seed (b) Third seed

Figure 8.29: Results on the evaluation dataset for the models without embedder fine-tuning when varying the learning rate.



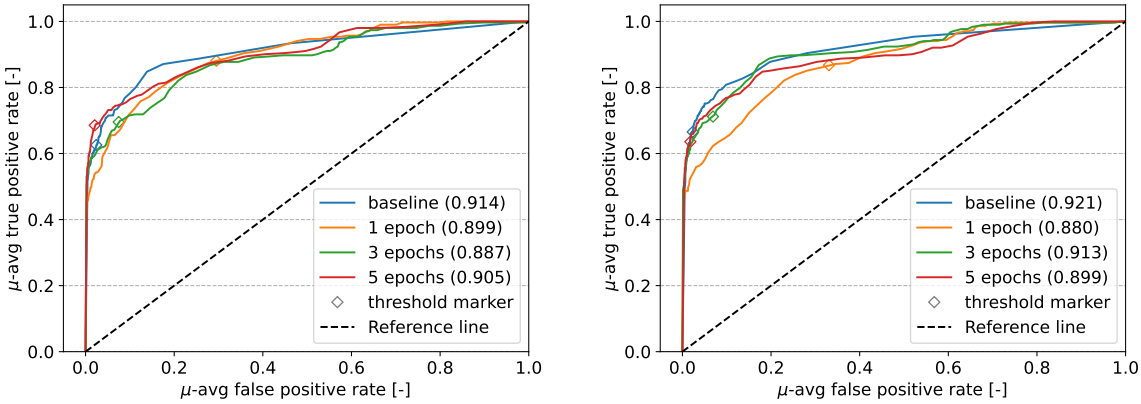
(a) Second seed (b) Third seed

Figure 8.30: Loss curves for the *baseline* on other datasets.



(a) Second seed (b) Third seed

Figure 8.31: Evaluation dataset results for *baseline* trained with larger datasets.



(a) Second seed

(b) Third seed

Figure 8.32: Evaluation dataset results for *baseline* with limited number of epochs.

Bibliography

[1] G. Derval, A. Gego, P. Reinbold, B. Frantzen, and P. Van Roy, "Automatic grading of programming exercises in a MOOC using the inginious platform," in *Proceedings of the 2015 European Stakeholder Summit on experiences and best practices in and around MOOCs*, Mons, Belgium, May 2015, pp. 86–91. [Online]. Available: https://www.researchgate.net/publication/340817309_Potentiating_the_human_dimension_in_Language_MOOCs

[2] M. Pradel and K. Sen, "DeepBugs: a learning approach to name-based bug detection," *Proceedings of the ACM on Programming Languages*, vol. 2, pp. 147:1–147:25, Oct. 2018. [Online]. Available: https://dl.acm.org/doi/10.1145/3276517

[3] M. Sundararajan, A. Taly, and Q. Yan, "Axiomatic attribution for deep networks," in *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ser. ICML'17. Sydney, NSW, Australia: JMLR.org, Aug. 2017, pp. 3319–3328.

[4] R. Swartz, "Mistakes as an Important Part of the Learning Process," *The High School Journal*, vol. 59, no. 6, pp. 246–257, 1976, publisher: University of North Carolina Press. [Online]. Available: https://www.jstor.org/stable/40365921

[5] M. Kapur and K. Bielaczyc, "Designing for Productive Failure," *Journal of the Learning Sciences*, vol. 21, no. 1, pp. 45–83, Jan. 2012, publisher: Routledge. [Online]. Available: https://doi.org/10.1080/10508406.2011.591717

[6] D. Boud and E. Molloy, "Feedback in higher and professional education," *Understanding It and Doing It Well*, 2013.

[7] M. Tulis, "Error Management Behavior in Classrooms: Teachers' Responses to Student Mistakes," *Teaching and Teacher Education: An International Journal of Research and Studies*, vol. 33, pp. 56–68, Jul. 2013, publisher: Elsevier ERIC Number: EJ1003550.

[8] J. Johnson, M. Panagioti, J. Bass, L. Ramsey, and R. Harrison, "Resilience to emotional distress in response to failure, error or mistakes: A systematic review," *Clinical psychology review*, vol. 52, pp. 19–42, Mar. 2017. [Online]. Available: https://doi.org/10.1016/j.cpr.2016.11.007

[9] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan *et al.*, "Language Models are Few-Shot Learners," in *Advances in Neural Information Processing Systems*, vol. 33. Curran Associates, Inc., 2020, pp. 1877–1901. [Online]. Available: https://papers.nips.cc/paper/2020/hash/1457c0d6bfcb4967418bfb8ac142f64a-Abstract.html

[10] P. P. Ray, "ChatGPT: A comprehensive review on background, applications, key challenges, bias, ethics, limitations and future scope," *Internet of Things and Cyber-Physical Systems*, vol. 3, pp. 121–154, Jan. 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S266734522300024X

[11] T. Wu, S. He, J. Liu *et al.*, "A Brief Overview of ChatGPT: The History, Status Quo and Potential Future Development," *IEEE/CAA Journal of Automatica Sinica*, vol. 10, no. 5, pp. 1122–1136, May 2023, conference Name: IEEE/CAA Journal of Automatica Sinica. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/10113601

[12] A. Haleem, M. Javaid, and R. P. Singh, "An era of ChatGPT as a significant futuristic support tool: A study on features, abilities, and challenges," *BenchCouncil Transactions on Benchmarks, Standards and Evaluations*, vol. 2, no. 4, p. 100089, Oct. 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2772485923000066

[13] S. S. Biswas, "Role of ChatGPT in Public Health," *Annals of Biomedical Engineering*, vol. 51, no. 5, pp. 868–869, May 2023. [Online]. Available: https://doi.org/10.1007/s10439-023-03172-7

[14] A. Umbraško and R. Drury, "Applying ChatGPT in AI-based dynamic video game narrative generation system," Oct. 2023. [Online]. Available: https://gupea.ub.gu.se/handle/2077/78909

[15] M. K. Audichya and J. R. Saini, "ChatGPT for Creative Writing and Natural Language Generation in Poetry and Prose," in *2023 International Conference on Advanced Computing Technologies and Applications (ICACTA)*, Oct. 2023, pp. 1–7. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/10392805

[16] Y. Liu and M. Lapata, "Text Summarization with Pretrained Encoders," in *Proceedings of the 2019 Conference on Empirical Methods in Natural Language*

Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP), K. Inui, J. Jiang, V. Ng, and X. Wan, Eds. Hong Kong, China: Association for Computational Linguistics, Nov. 2019, pp. 3730–3740. [Online]. Available: https://aclanthology.org/D19-1387

[17] Z. Akkalyoncu Yilmaz, S. Wang, W. Yang, H. Zhang, and J. Lin, "Applying BERT to Document Retrieval with Birch," in *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP): System Demonstrations*, S. Padó and R. Huang, Eds. Hong Kong, China: Association for Computational Linguistics, Nov. 2019, pp. 19–24. [Online]. Available: https://aclanthology.org/D19-3004

[18] M. E. Peters, M. Neumann, L. Zettlemoyer, and W.-t. Yih, "Dissecting Contextual Word Embeddings: Architecture and Representation," in *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, E. Riloff, D. Chiang, J. Hockenmaier, and J. Tsujii, Eds. Brussels, Belgium: Association for Computational Linguistics, Oct. 2018, pp. 1499–1509. [Online]. Available: https://aclanthology.org/D18-1179

[19] A. Miaschi and F. Dell'Orletta, "Contextual and Non-Contextual Word Embeddings: an in-depth Linguistic Investigation," in *Proceedings of the 5th Workshop on Representation Learning for NLP*, S. Gella, J. Welbl, M. Rei, F. Petroni, P. Lewis, E. Strubell, M. Seo, and H. Hajishirzi, Eds. Association for Computational Linguistics, Jul. 2020, pp. 110–119. [Online]. Available: https://aclanthology.org/2020.repl4nlp-1.15

[20] Q. Liu, M. J. Kusner, and P. Blunsom, "A Survey on Contextual Embeddings," Mar. 2020. [Online]. Available: https://ui.adsabs.harvard.edu/abs/2020arXiv200307278L

[21] S. K. Singh, S. Kumar, and P. S. Mehra, "Chat GPT & Google Bard AI: A Review," in *2023 International Conference on IoT, Communication and Automation Technology (ICICAT)*, Jun. 2023, pp. 1–6. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/10263706

[22] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, J. Burstein, C. Doran, and T. Solorio, Eds. Minneapolis, Minnesota: Association for Computational Linguistics, Jun. 2019, pp. 4171–4186. [Online]. Available: https://aclanthology.org/N19-1423

[23] Y. Jia and M. Harman, "An Analysis and Survey of the Development of Mutation Testing," *IEEE Transactions on Software Engineering*, vol. 37, no. 5, pp. 649–678, Sep. 2011. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/5487526

[24] C. Douce, D. Livingstone, and J. Orwell, "Automatic test-based assessment of programming: A review," *Journal on Educational Resources in Computing*, vol. 5, no. 3, pp. 4–es, Sep. 2005. [Online]. Available: https://doi.org/10.1145/1163405.1163409

[25] R. Romli, S. Sulaiman, and K. Z. Zamli, "Automatic programming assessment and test data generation a review on its approaches," in *2010 International Symposium on Information Technology*, vol. 3, Jun. 2010, pp. 1186–1192, iSSN: 2155-899X. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/5561488

[26] H. Keuning, J. Jeuring, and B. Heeren, "A Systematic Literature Review of Automated Feedback Generation for Programming Exercises," *ACM Transactions on Computing Education*, vol. 19, no. 1, pp. 3:1–3:43, Sep. 2018. [Online]. Available: https://doi.org/10.1145/3231711

[27] J. C. Paiva, J. P. Leal, and A. Figueira, "Automated Assessment in Computer Science Education: A State-of-the-Art Review," *ACM Transactions on Computing Education*, vol. 22, no. 3, pp. 34:1–34:40, Jun. 2022. [Online]. Available: https://doi.org/10.1145/3513140

[28] M. Messer, N. C. C. Brown, M. Kölling, and M. Shi, "Automated Grading and Feedback Tools for Programming Education: A Systematic Review," *ACM Transactions on Computing Education*, vol. 24, no. 1, pp. 1–43, Mar. 2024, arXiv:2306.11722 [cs]. [Online]. Available: http://arxiv.org/abs/2306.11722

[29] S. H. Edwards and M. A. Perez-Quinones, "Web-CAT: automatically grading programming assignments," in *Proceedings of the 13th annual conference on Innovation and technology in computer science education*, ser. ITiCSE '08. New York, NY, USA: Association for Computing Machinery, Jun. 2008, p. 328. [Online]. Available: https://doi.org/10.1145/1384271.1384371

[30] M. Striewe, "An architecture for modular grading and feedback generation for complex exercises," *Science of Computer Programming*, vol. 129, pp. 35–47, Nov. 2016. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167642316300260

[31] D. Croft and M. England, "Computing with CodeRunner at Coventry University: Automated summative assessment of Python and C++ code,"

in *Proceedings of the 4th Conference on Computing Education Practice 2020*, Jan. 2020, pp. 1–4, arXiv:1911.11085 [cs]. [Online]. Available: http://arxiv.org/abs/1911.11085

[32] T. Staubitz, H. Klement, R. Teusner, J. Renz, and C. Meinel, "CodeOcean - A versatile platform for practical programming excercises in online environments," in *2016 IEEE Global Engineering Education Conference (EDUCON)*, Apr. 2016, pp. 314–323, iSSN: 2165-9567. [Online]. Available: https://ieeexplore.ieee.org/document/7474573

[33] M. Brito and C. Gonçalves, "Codeflex: A Web-based Platform for Competitive Programming," in *2019 14th Iberian Conference on Information Systems and Technologies (CISTI)*, Jun. 2019, pp. 1–6, iSSN: 2166-0727. [Online]. Available: https://ieeexplore.ieee.org/document/8760776

[34] G. Benetti, G. Roveda, D. Giuffrida, and T. Facchinetti, "Coderiu: a cloud platform for computer programming e-learning," in *2019 IEEE 17th International Conference on Industrial Informatics (INDIN)*, vol. 1, Jul. 2019, pp. 1126–1132, iSSN: 2378-363X. [Online]. Available: https://ieeexplore.ieee.org/document/8972058

[35] B. Boe, C. Hill, M. Len, G. Dreschler, P. Conrad, and D. Franklin, "Hairball: lint-inspired static analysis of scratch projects," in *Proceeding of the 44th ACM technical symposium on Computer science education*, ser. SIGCSE '13. New York, NY, USA: Association for Computing Machinery, Mar. 2013, pp. 215–220. [Online]. Available: https://doi.org/10.1145/2445196.2445265

[36] C. G. von Wangenheim, J. C. R. Hauck *et al.*, "CodeMaster–Automatic Assessment and Grading of App Inventor and Snap! Programs," *Informatics in Education*, vol. 17, no. 1, pp. 117–150, 2018, publisher: Vilnius University Institute of Mathematics and Informatics, Lithuanian Academy of Sciences ERIC Number: EJ1177148. [Online]. Available: https://eric.ed.gov/?id=EJ1177148

[37] M. Talbot, K. Geldreich, J. Sommer, and P. Hubwieser, "Re-use of programming patterns or problem solving? representation of scratch programs by TGraphs to support static code analysis," in *Proceedings of the 15th Workshop on Primary and Secondary Computing Education*, ser. WiPSCE '20. New York, NY, USA: Association for Computing Machinery, Oct. 2020, pp. 1–10. [Online]. Available: https://doi.org/10.1145/3421590.3421604

[38] G. Haldeman, A. Tjang, M. Babeş-Vroman *et al.*, "Providing Meaningful Feedback for Autograding of Programming Assignments," in *Proceedings of the*

49th ACM Technical Symposium on Computer Science Education, ser. SIGCSE '18. New York, NY, USA: Association for Computing Machinery, Feb. 2018, pp. 278–283. [Online]. Available: https://doi.org/10.1145/3159450.3159502

[39] A. Head, E. Glassman, G. Soares *et al.*, "Writing Reusable Code Feedback at Scale with Mixed-Initiative Program Synthesis," in *Proceedings of the Fourth (2017) ACM Conference on Learning @ Scale*, ser. L@S '17. New York, NY, USA: Association for Computing Machinery, Apr. 2017, pp. 89–98. [Online]. Available: https://dl.acm.org/doi/10.1145/3051457.3051467

[40] R. Singh, S. Gulwani, and A. Solar-Lezama, "Automated feedback generation for introductory programming assignments," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '13. New York, NY, USA: Association for Computing Machinery, Jun. 2013, pp. 15–26. [Online]. Available: https://doi.org/10.1145/2491956.2462195

[41] H. Keuning, B. Heeren, and J. Jeuring, "Strategy-based feedback in a programming tutor," in *Proceedings of the Computer Science Education Research Conference*, ser. CSERC '14. New York, NY, USA: Association for Computing Machinery, Nov. 2014, pp. 43–54. [Online]. Available: https://doi.org/10.1145/2691352.2691356

[42] K. Rivers and K. R. Koedinger, "Data-Driven Hint Generation in Vast Solution Spaces: a Self-Improving Python Programming Tutor," *International Journal of Artificial Intelligence in Education*, vol. 27, no. 1, pp. 37–64, Mar. 2017. [Online]. Available: https://doi.org/10.1007/s40593-015-0070-z

[43] K. Mens, S. Nijssen, and H.-S. Pham, "The good, the bad, and the ugly: mining for patterns in student source code," in *Proceedings of the 3rd International Workshop on Education through Advanced Software Engineering and Artificial Intelligence*, ser. EASEAI 2021. New York, NY, USA: Association for Computing Machinery, Aug. 2021, pp. 1–8. [Online]. Available: https://doi.org/10.1145/3472673.3473958

[44] J. Liénard, K. Mens, and S. Nijssen, "Extracting Unit Tests from Patterns Mined in Student Code to Provide Improved Feedback in Autograders."

[45] D. Di Nucci, H.-S. Pham, J. Fabry *et al.*, "A Language-Parametric Modular Framework for Mining Idiomatic Code Patterns," 2019. [Online]. Available: https://www.semanticscholar.org/paper/A-Language-Parametric-Modular-Framework-for-Mining-Nucci-Pham/6c2a8921e820bc42ac9bcc02691282c8bc43f535

[46] H. S. Pham, S. Nijssen, K. Mens *et al.*, "Mining Patterns in Source Code Using Tree Mining Algorithms," in *Discovery Science*, ser. Lecture Notes in Computer Science, P. Kralj Novak, T. Šmuc, and S. Džeroski, Eds. Cham: Springer International Publishing, 2019, pp. 471–480.

[47] T. Ishio, H. Date, T. Miyake, and K. Inoue, "Mining Coding Patterns to Detect Crosscutting Concerns in Java Programs," in *2008 15th Working Conference on Reverse Engineering*, Oct. 2008, pp. 123–132, iSSN: 2375-5369. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/4656401

[48] H. Kagdi, M. L. Collard, and J. I. Maletic, "Comparing Approaches to Mining Source Code for Call-Usage Patterns," in *Fourth International Workshop on Mining Software Repositories (MSR'07:ICSE Workshops 2007)*, May 2007, pp. 20–20, iSSN: 2160-1860. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/4228657

[49] W. Qu, Y. Jia, and M. Jiang, "Pattern mining of cloned codes in software systems," *Information Sciences*, vol. Complete, no. 259, pp. 544–554, 2014. [Online]. Available: https://www.infona.pl//resource/bwmeta1.element.elsevier-674a00a7-1190-35c3-8892-e303358a2de7

[50] Z. Balanyi and R. Ferenc, "Mining design patterns from C++ source code," in *International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings.*, Sep. 2003, pp. 305–314, iSSN: 1063-6773. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/1235436

[51] J. Dong, Y. Zhao, and T. Peng, "A review of design pattern mining techniques," *International Journal of Software Engineering and Knowledge Engineering*, vol. 19, no. 06, pp. 823–855, Sep. 2009, publisher: World Scientific Publishing Co. [Online]. Available: https://www.worldscientific.com/doi/abs/10.1142/S021819400900443X

[52] S. Chakraborty, R. Tomsett, R. Raghavendra *et al.*, "Interpretability of deep learning models: A survey of results," in *2017 IEEE SmartWorld, Ubiquitous Intelligence & Computing, Advanced & Trusted Computed, Scalable Computing & Communications, Cloud & Big Data Computing, Internet of People and Smart City Innovation (SmartWorld/SCALCOM/UIC/ATC/CBDCom/IOP/SCI)*, Aug. 2017, pp. 1–6. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8397411

[53] Y. Zhang, P. Tiňo, A. Leonardis, and K. Tang, "A Survey on Neural Network Interpretability," *IEEE Transactions on Emerging Topics in Computational Intelligence*, vol. 5, no. 5, pp. 726–742, Oct. 2021, conference Name: IEEE

Transactions on Emerging Topics in Computational Intelligence. [Online]. Available: https://ieeexplore.ieee.org/document/9521221

[54] X. Li, H. Xiong, X. Li *et al.*, "Interpretable deep learning: interpretation, interpretability, trustworthiness, and beyond," *Knowledge and Information Systems*, vol. 64, no. 12, pp. 3197–3234, Dec. 2022. [Online]. Available: https://doi.org/10.1007/s10115-022-01756-8

[55] K. Leino, S. Sen, A. Datta *et al.*, "Influence-Directed Explanations for Deep Convolutional Networks," in *2018 IEEE International Test Conference (ITC)*, Oct. 2018, pp. 1–8, iSSN: 2378-2250. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8624792

[56] K. Dhamdhere, M. Sundararajan, and Q. Yan, "How Important Is a Neuron?" May 2018. [Online]. Available: https://ui.adsabs.harvard.edu/abs/2018arXiv180512233D

[57] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient Estimation of Word Representations in Vector Space," Jan. 2013. [Online]. Available: https://ui.adsabs.harvard.edu/abs/2013arXiv1301.3781M

[58] R.-M. Karampatsis and C. Sutton, "SCELMo: Source Code Embeddings from Language Models," Apr. 2020, arXiv:2004.13214 [cs]. [Online]. Available: http://arxiv.org/abs/2004.13214

[59] M. E. Peters, M. Neumann, M. Iyyer, M. Gardner, C. Clark, K. Lee, and L. Zettlemoyer, "Deep Contextualized Word Representations," in *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, M. Walker, H. Ji, and A. Stent, Eds. New Orleans, Louisiana: Association for Computational Linguistics, Jun. 2018, pp. 2227–2237. [Online]. Available: https://aclanthology.org/N18-1202

[60] V. Efstathiou and D. Spinellis, "Semantic Source Code Models Using Identifier Embeddings," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, May 2019, pp. 29–33, iSSN: 2574-3864. [Online]. Available: https://ieeexplore.ieee.org/document/8816775

[61] T. Ben-Nun, A. S. Jakobovits, and T. Hoefler, "Neural code comprehension: a learnable representation of code semantics," in *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, ser. NIPS'18. Red Hook, NY, USA: Curran Associates Inc., Dec. 2018, pp. 3589–3601.

[62] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: learning distributed representations of code," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 40:1–40:29, Jan. 2019. [Online]. Available: https://dl.acm.org/doi/10.1145/3290353

[63] ——, "A general path-based representation for predicting program properties," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2018. New York, NY, USA: Association for Computing Machinery, Jun. 2018, pp. 404–419. [Online]. Available: https://doi.org/10.1145/3192366.3192412

[64] D. Bahdanau, K. Cho, and Y. Bengio, "Neural Machine Translation by Jointly Learning to Align and Translate," Sep. 2014. [Online]. Available: https://ui.adsabs.harvard.edu/abs/2014arXiv1409.0473B

[65] U. Alon, S. Brody, O. Levy, and E. Yahav, "code2seq: Generating Sequences from Structured Representations of Code," Sep. 2018. [Online]. Available: https://openreview.net/forum?id=H1gKYo09tX&utm_campaign=NLP%20News&utm_medium=email&utm_source=Revue%20newsletter

[66] Y. Shi, K. Shah, W. Wang, S. Marwan, P. Penmetsa, and T. Price, "Toward Semi-Automatic Misconception Discovery Using Code Embeddings," in *LAK21: 11th International Learning Analytics and Knowledge Conference*, ser. LAK21. New York, NY, USA: Association for Computing Machinery, Apr. 2021, pp. 606–612. [Online]. Available: https://doi.org/10.1145/3448139.3448205

[67] A. Vaswani, N. Shazeer, N. Parmar *et al.*, "Attention is All you Need," in *Advances in Neural Information Processing Systems*, vol. 30. Curran Associates, Inc., 2017. [Online]. Available: https://papers.nips.cc/paper_files/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html

[68] A. Neelakantan, T. Xu, R. Puri *et al.*, "Text and Code Embeddings by Contrastive Pre-Training," Jan. 2022, arXiv:2201.10005 [cs]. [Online]. Available: http://arxiv.org/abs/2201.10005

[69] Z. Chen and M. Monperrus, "A Literature Study of Embeddings on Source Code," Apr. 2019, arXiv:1904.03061 [cs, stat]. [Online]. Available: http://arxiv.org/abs/1904.03061

[70] A. Kanade, P. Maniatis, G. Balakrishnan, and K. Shi, "Learning and evaluating contextual embedding of source code," in *Proceedings of the 37th International Conference on Machine Learning*, ser. ICML'20, vol. 119. JMLR.org, Jul. 2020, pp. 5110–5121.

[71] Z. Feng, D. Guo, D. Tang, N. Duan *et al.*, "CodeBERT: A Pre-Trained Model for Programming and Natural Languages," in *Findings of the Association for Computational Linguistics: EMNLP 2020*. Online: Association for Computational Linguistics, Nov. 2020, pp. 1536–1547. [Online]. Available: https://aclanthology.org/2020.findings-emnlp.139

[72] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu *et al.*, "GraphCodeBERT: Pre-training Code Representations with Data Flow," Sep. 2020. [Online]. Available: https://ui.adsabs.harvard.edu/abs/2020arXiv200908366G

[73] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, May 2015, publisher: Nature Publishing Group. [Online]. Available: https://www.nature.com/articles/nature14539

[74] J. E. Hopcroft, R. Motwani, and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. USA: Addison-Wesley Longman Publishing Co., Inc., 2006.

[75] R. Sadre, "Languages and translators course," the CFG definition we used follows the notations developed during this UCLouvain course. [Online]. Available: https://sites.uclouvain.be/archives-portail/cdc2022/en-cours-2022-linfo2132

[76] J. Jones, "Abstract syntax tree implementation idioms," *Pattern Languages of Program Design*, 2003, proceedings of the 10th Conference on Pattern Languages of Programs (PLoP2003). [Online]. Available: http://hillside.net/plop/plop2003/Papers/Jones-ImplementingASTs.pdf

[77] P. Dupont, "Computational linguistics course," the explanations about the attention, embeddings, Transformers and BERT has been supported by the lectures of this course. [Online]. Available: https://uclouvain.be/en-cours-2023-linfo2263

[78] D. Jurafsky and J. H. Martin, *Speech and Language Processing, 3rd edition (draft)*. [Online]. Available: https://web.stanford.edu/~jurafsky/slp3/

[79] Y. Wu, M. Schuster, Z. Chen *et al.*, "Google's neural machine translation system: Bridging the gap between human and machine translation," *CoRR*, vol. abs/1609.08144, 2016. [Online]. Available: http://arxiv.org/abs/1609.08144

[80] L. Zhuang, L. Wayne, S. Ya, and Z. Jun, "A robustly optimized BERT pre-training approach with post-training," in *Proceedings of the 20th Chinese National Conference on Computational Linguistics*, S. Li, M. Sun, Y. Liu, H. Wu, K. Liu, W. Che, S. He, and G. Rao, Eds. Huhhot, China: Chinese

Information Processing Society of China, Aug. 2021, pp. 1218–1227. [Online]. Available: https://aclanthology.org/2021.ccl-1.108

[81] A. Shrikumar, J. Su, and A. Kundaje, "Computationally efficient measures of internal neuron importance," *CoRR*, vol. abs/1807.09946, 2018. [Online]. Available: http://arxiv.org/abs/1807.09946

[82] N. Kokhlikyan, V. Miglani, M. Martin *et al.*, "Captum: A unified and generic model interpretability library for pytorch," *CoRR*, vol. abs/2009.07896, 2020. [Online]. Available: https://arxiv.org/abs/2009.07896

[83] A. Paszke, S. Gross, F. Massa *et al.*, *PyTorch: an imperative style, high-performance deep learning library.* Red Hook, NY, USA: Curran Associates Inc., 2019.

[84] T. Wolf, L. Debut, V. Sanh *et al.*, "Transformers: State-of-the-art natural language processing," in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, Q. Liu and D. Schlangen, Eds. Online: Association for Computational Linguistics, Oct. 2020, pp. 38–45. [Online]. Available: https://aclanthology.org/2020.emnlp-demos.6

[85] L. Chiodini, I. Moreno Santos, A. Gallidabino, A. Tafliovich, A. L. Santos, and M. Hauswirth, "A Curated Inventory of Programming Language Misconceptions," in *Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education V. 1*, ser. ITiCSE '21. New York, NY, USA: Association for Computing Machinery, Jun. 2021, pp. 380–386. [Online]. Available: https://doi.org/10.1145/3430665.3456343

[86] M. Gardner, J. Grus, M. Neumann *et al.*, "Allennlp: A deep semantic natural language processing platform," 2017.

[87] B. Courty, V. Schmidt, S. Luccioni *et al.*, "mlco2/codecarbon: v2.4.1," May 2024. [Online]. Available: https://doi.org/10.5281/zenodo.11171501

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**