Feedback with BERT: When Detecting Students' Misconceptions Becomes Automatic

Guillaume Steveny[1,2], Julien Lienard[1], Kim Mens[1], and Siegfried Nijssen[1]

[1] ICTEAM institute, Université catholique de Louvain (UCLouvain), Belgium
{firstname}.{lastname}@uclouvain.be
[2] Former student, new email: gsteveny.research@gmail.com

Abstract. When learning a new programming language, students can benefit from feedback on their work, to get a better overview of their level, strengths and shortcomings. However, giving personalised feedback on misconceptions is complex due to a lack of means or resources; the design of automated tests and feedback is cumbersome and time-consuming. Our work aims to overcome some of these limitations by enabling automatic feedback thanks to a machine learning model. We developed a multi-label classification architecture following latest advances in natural language processing. By using code embeddings, i.e. generated vectors on students' code submissions, our system allows to detect specific misconceptions occurring in code snippets, and provide predefined feedback based on these classes. To control the classes and enable the training of our deep neural network, we developed an approach inspired by DeepBugs. 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 highlight limits of our current mutation labelling technique and improvements to be conducted as further work.

Keywords: Machine learning · Feedback · Programming language · Python · Embeddings · Mutation · Programming Misconceptions.

1 Introduction

Context. Numerous universities offer programming language courses. Our university does not depart from this practice by integrating, in the first Bachelor years of both engineering and computer science, an introductory programming course using the Python language. Students following it have to execute different tasks to certify their knowledge. They read a syllabus teaching the basics of programming, attend theory lessons covering a reference book, participate in tutored exercise sessions, and produce weekly coding assignments where they solve more abstract problems based on the material taught during the week.

In this teaching process, two mechanisms enable students to receive feedback on their work: via their tutor or via the course's autograding platform. Tutors

meet their students two hours per week and can provide brief global comments on their code submissions. The online grading platform plays the role of an online teaching assistant that provides instantaneous feedback on coding assignments. It enables deploying environments that test student submissions automatically.

Problem. Each of these feedback mechanisms has its flaws. Tutors can only devote a limited amount of time to students. Their follow-up does not always match the students' needs and expectations. The autograding platform, on the other hand, offers unit tests that check the correct behaviour of code or, to a lesser extent, the structure of student-produced code, but does not offer help in understanding misconceptions. While manual coding of additional feedback in an autograding platform is possible in principle, writing tests that highlight mistakes or possible misconceptions is difficult and time-consuming. Inaccurate feedback could be detrimental to the students' learning process, making it even harder to implement such tests and limiting their deployment on most exercises.

Objective. The approach that we study in this paper is how we can use Machine Learning techniques (ML), and Natural Language Processing (NLP) tools in particular, to provide feedback to students with respect to programming misconceptions present in their code. We wish to do so in a responsible manner.

Approach. Our work targets the creation of a code classification system based on classical programming misconceptions students exhibit, like for example using a `print` statement where instead they should have used a `return`. Hence, we strictly restrict the type of misconceptions identified by the system. Every program written by a student, once processed by the model, will obtain a prediction associated with a confidence level, triggering meaningful, predefined feedback to the students to help them identify their potential misconceptions.

In this paper we treat the problem of identifying programming misconceptions as a multi-label classification problem. Most architectures recently deployed to generate text summaries [17] or for information retrieval tasks [1] rely on *embeddings* capturing the semantic essence of a language [16,18,22]. We rely on the BERT architecture [7], building on code embeddings generated by the GraphCodeBERT model [11], developed by Microsoft Research. An important challenge that we tackle in this work is that the amount of data available in this application context is too limited for training a language model. To train a classifier to recognise the desired misconceptions, we propose a process in which a large number of labelled code mutations are generated from students' code submissions. By using such a data-oriented approach, we aim to train models that show predictable behaviour. Finally, we argue for the use of explainable AI techniques to help students understand the prediction of the ML model.

As future work we envision integrating the output of our model inside an autograder, allowing a continuous learning approach where the students and teaching staff's feedback is accounted for to improve the model performance.

2 Related Work

Automated feedback generation. Multiple systems have been deployed previously to generate feedback for students automatically. The review by Paiva et al. [20] highlights that most tools rely on output comparison [6,8,29] or unit testing [2,4,27]. Other approaches are based on static analysis to identify bad coding practices of students and remedy these [3,31,32]. Each of these systems require additional work to be deployed for each exercise.

The strategy selected for our work will limit the workload on the teaching staff while providing a large range of detectable aspects, such as bad coding practices. Rather than providing a list of modifications to be applied by the students [15,24,26], the model will limit itself to the classification of such instances, allowing the teaching staff to adapt the feedback according to the needs of the students: syllabus references, error indications or hints for possible corrections.

Code embeddings. Different mechanisms are already using code embeddings for bug classification. DeepBugs [23], which served as inspiration for our mutation labelling process described later, uses Word2Vec [19] static embeddings on JavaScript source code. SCELMo [14] goes a step further by using ELMo as an embedder. Our approach differentiates from these two systems by using a BERT-like model [7], benefiting from the transfer learning abilities of this architecture, and performing the classification on Python source code. The mutations we could inject are also representative of students' misconceptions rather than bugs for code repair in a more professional context.

CuBERT [13] also relies on injected artificial bugs for classification. However, this model contains a BERT-large configuration, which could consume too many resources to be deployed efficiently during the semester. Furthermore, this approach trains a separate model for each type of injected bug, while our approach trains one model, thus leading to a more responsible usage of resources. Finally, their mutations are predefined, while ours can be configured thanks to user-defined rule sets.

Based on the Code2Vec model, Shi et al. [25] constructed a procedure to identify student misconceptions. By using code embeddings, the researchers tried to group snippets into clusters. These constituted a basis for labelling the student submissions afterwards. However, their approach is only 'semi-automatic', requiring humans in the loop to identify misconceptions from the clusters and discard non-meaningful groupings. Our current strategy could be described as fully automatic since, once the mutations corresponding to misconceptions have been designed and used for training, the model could predict these without further processing by the teaching staff.

3 Background

In this section, we briefly present GraphCodeBERT since it is the central part of the classification architecture we chose. As our approach targets multi-label instances, we also explain the loss metric we selected to train the model.

GraphCodeBERT. A successor of CodeBERT [9], GraphCodeBERT [11] targets creating a pre-trained model containing information about six programming languages: Python, Java, Go, JavaScript, PHP and Ruby. It is based on the RoBERTa-base architecture [34] and requires three components as input: `token ids`, `attention mask` and `positional ids`. The `positional ids` sequence is an ordered sequence of integers representing the token positions.

A model input contains three specific segments: text, code and a DataFlow Graph (DFG). The main component the model requires is the code. One could prepend a text segment before the code one, which corresponds to the docstrings extracted from the code inputs. This part can be omitted when the model should focus on code aspects rather than on text-code pairs. The DFG is a specific addition of GraphCodeBERT. It contains information about the constants and variables used inside the submitted code. This structure omits unassigned or unused variables or constants. To translate it into RoBERTa inputs, the authors created a strategy to represent the nodes and the edges inside the `attention mask`: graph-guided masked attention. Further details are available in the original paper by the Guo et al. [11].

Loss metric. Using the classical Cross-Entropy loss in the context of a multi-label classification will push each instance towards one specific label but not all its associated labels. 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 1, should push the instances towards all the labels that are associated with them.

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

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.

4 Approach

This section provides the information necessary to understand the classification architecture we selected to work with code embeddings. We will then present our strategy, inspired by the DeepBugs system [23], to create the training datasets.

4.1 Classification model

Figure 1 depicts a schematic representation of the architecture we configured.

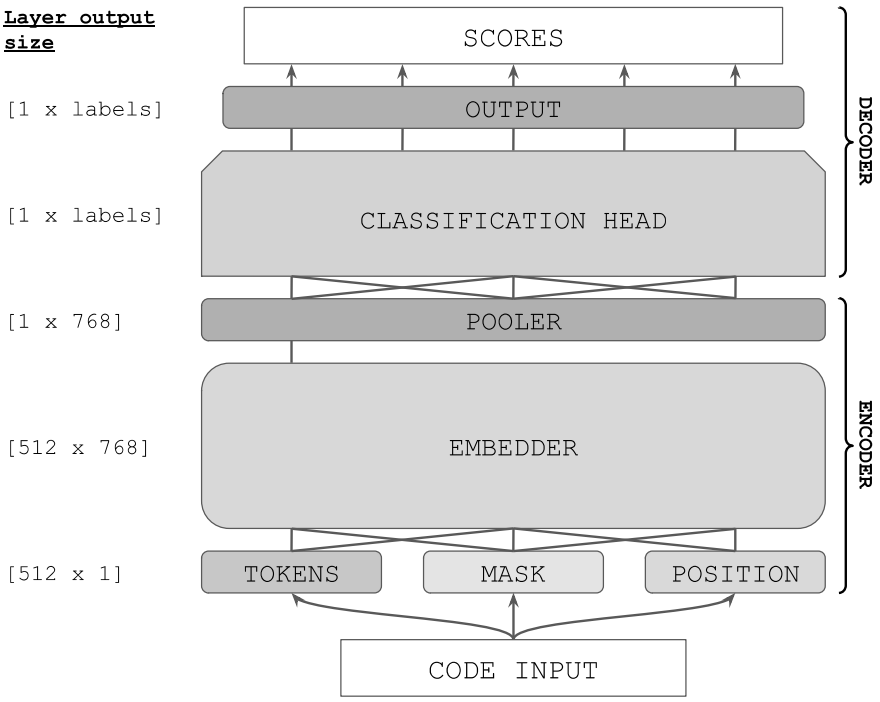


Fig. 1: Schematic representation of the components of our classification model.

Structure. The model provides a classification architecture similar to models proposed for sentiment analysis tasks. The instances are transformed into three parts: the `token ids`, the `attention mask` and the `positional ids`.

We decided to truncate each code to the maximal window length of the embedder, i.e. 512 tokens. We prefered this strategy over splitting a code snippet into multiple inputs, in order to ensure dependencies inside the snippet not to be disrupted. For example, if a variable is only defined at the start of the snippet, the other windows we could construct would miss this assignment, leading to incorrect class predictions. This strategy led to 0.777% of tokens lost for the training and evaluation datasets, i.e. 0.051% of the inputs are lost on average.

These three tensors are given to the pre-trained GraphCodeBERT model to generate, for each input token, a 768-dimensional vector. However, for our classification procedure, we only need one 768-dimensional embedding to represent the entire student submission. This embedding extraction is the role of the "`pooler`". The corresponding 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 default parameters are one additional layer with 768 hidden units and LeakyReLU as activation.

The `output` layer normalises the `logits` to assign a score to each possible class. Each value will represent the model confidence in predicting this label. This computation is a *sigmoid* applied on the `logits`. Here, the scores should not sum to one since we target multi-label tasks.

As depicted in Figure 1, 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.

Implementation. To implement such an architecture, we used the AllenNLP [10] and HuggingFace [33] libraries along with PyTorch [21]. We decided to use the "GraphCodeBERT-py" pre-trained model from Enoch (Ensheng Shi, PhD student at Xi'an Jiaotong University) [25] available on the HuggingFace's models storage. The weights have been trained on Python source code inside the CodeSearchNet dataset [12]. Every parameter of the model architecture, like the classification head or the embedder, can be configured using yaml files.

4.2 Mutation labelling

To train the previously described architecture, since we placed ourselves in a supervised approach, this process would require labelled instances. Using bugged source code found online from more experienced programmers is expected to limit the model understanding of the simpler students' problems, like misusing a `print` for a `return`. Manually labelling more than ten thousand failed submissions from students to the course exercises would have been highly time-consuming and required to be consistent during the entire process. We would like to have a strict control over the situations in which misconceptions are identified.

We propose here to study an idea previously presented in the DeepBugs system [23]. 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 that is consistent and controlled. 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 incorrect.

We developed a 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 applies these to the original instances to modify them as desired. To write the rules while providing freedom to the user in their definition and extensive modification power, the program combines `Comby` and `RedBaron`.

`Comby`[3] is a tool to perform modifications to source code. It allows one to specify matching rules used for replacement. Its advantage relies on its ability to specify "holes" containing part of the match.

`RedBaron`[4] is a Python library relying 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.

The first tool allows the user to have the freedom to write simple mutation rules, like, for example, replacing `return` by `print` statements. The second will enable more powerful mutations that are part of a list of 41 predefined rules we designed, like, for example, removing the condition update of a while loop.

The mutation program we designed was used on submissions considered as correct by our autograding platform, for 23 programming assignments taken from the studied course during the first semester of the academic year 2023–2024. These cover simple questions about functions, lists, dictionaries and file handling with Python. This process resulted in 13804 submissions collected. We ensured each submission analysed was anonymised and without any comments, making it impossible to identify specific students. Our university allowed such use of data. After considering the selected exercise contexts, a manual analysis of students' errors and a theoretical exploration [5], we designed 15 rule sets covering 14 misconceptions or mistakes. These labels include using `print` instead of `return` (`print_return`), having incoherent loops (`bad_loop`) or forgetting to close a file (`bad_file`). Further details about the designed classes are provided in Appendix D of the master thesis on which this paper is based [28] and the final mutation rules can be found on the associated GitHub repository[5].

This list of misconceptions has been constructed throughout multiple iterations where misconceptions were added based on feedback received from the teaching staff. The set of detectable errors could be further augmented when new classes should be considered. However, drawbacks of doing so could include overloading students with too much information and increasing the risk of misclassifying instances due to a large output space. For this reason, we kept the number of classes limited.

5 Experiments

We designed two experiments to assess the mutation labelling procedure as well as our classification model. The first aims to train the model and assert its ability to understand the mutation classes. The second explores how the trained model performs on real students' submissions and how it could be improved.

[3] https://comby.dev/
[4] https://github.com/PyCQA/redbaron
[5] https://github.com/StevenGuyCap/CESReS-model/tree/main/mutation

5.1 Training task

Dataset. From the 13804 submissions we collected from semester exercises done by the students, we sampled 2500 programs for the mutation labelling procedure. These original instances allowed us to create 15176 multi-label mutants by using the mutation program. This dataset has been split into a training, a validation and a test set containing 81%, 9% and 10% of the instances, respectively. This split has been performed so that we could ensure that all mutants of an original submission were part of the same set.

Task. By using a multi-label soft-margin loss, the model is trained to determine the mutation classes we injected among the mutants. We relied on a baseline configuration using default parameters. The classification head contains a single additional layer, and all weights are learned using an AdamWOptimizer with a learning rate of 10^{-5}. We trained this configuration three times for 10 epochs with three different random seeds: 42, 33 and 18.

Results. By looking at the learning curve for the first seed (Figure 2), we can observe that the model has learned its task and reaches an optimal point around the 9^{th} epoch. The other seeds obtained the same behaviour. On the test set, the mean micro-average f1-score of 0.93716 with a standard deviation of 0.00209 indicates that the model extensively understood the mutations we injected inside the test instances without overfitting the training examples.

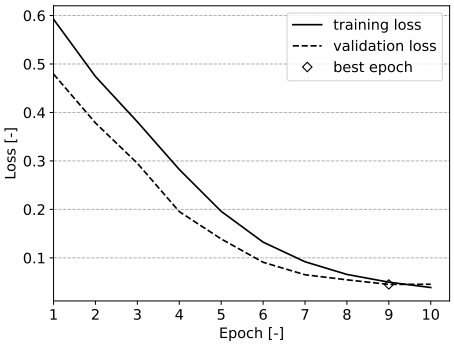


Fig. 2: Learning curve of the model for the first seed on the training task.

If we look at the label-wise f1-scores on the test set, there is some disparity between the classes. Table 1 summarises some cases we will highlight next.

`hardcoded_arg` is the easiest class for the model. This label relies on a specific mutation where one function argument is reassigned as the first instruction of the function's body. By looking at the two first lines of a submission, the model should already have enough information to handle this class.

Table 1: Average label-wise f1-scores the model obtained on the training task.

Label	Mean	Std
`hardcoded_arg`	0.99192	0.00131
`miss_try`	0.46154	0.00000
`miss_parenthesis`	0.40153	0.05098

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 the autograding platform 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 performing less for this label.

The `miss_parenthesis` label seems to be the most complex class to predict. This phenomenon relates to the mutation limitations our dataset contains. This class refers to the removal of one associative parenthesis block 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.

The second experiment will come back on these labels to identify how the model managed to detect these on real students' submissions.

5.2 Evaluation process

Dataset. 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 questions we selected for mutation, we will use (anonymised) submissions from the previous years' exams of the studied course. We chose this strategy for two 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. Secondly, the exam submissions do not overlap with our training set. The model should not have seen already the examples during its training process. This strategy will allow us to explore model generalisability.

After a random selection process, we obtained 161 submissions among the 121428 failed submissions collected from previous years. Using the 14 misconceptions classes we designed and the "`failed`" label to cover undecidable cases, we manually identified 302 label occurrences among the evaluation instances.

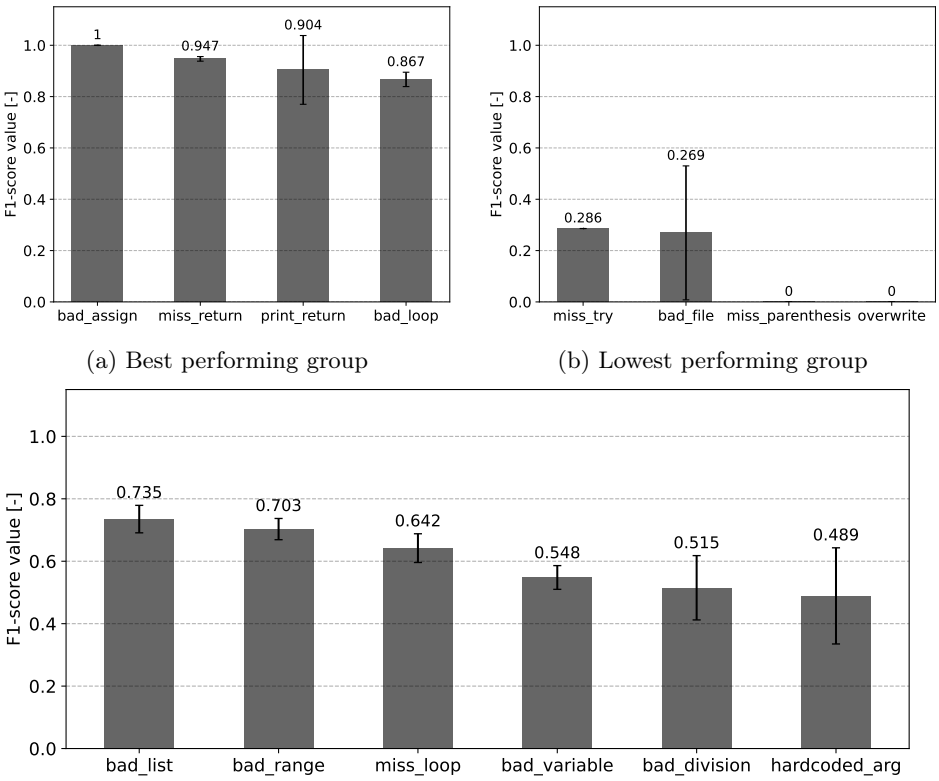
Process. Since we saved the weights of the configuration from the first experiment, we can load these and launch the evaluation process for them. Reusing these weights would spare computing and time resources. We measured each performance metric three times, one for each of the seeds from the previous experiment.

Results. Table 2 shows the performance of the model for each seed on the evaluation dataset. By considering the label-wise f1-scores of the model, we can highlight three groups. The first contains the four labels for which the mean f1-scores is above 80% (Figure 3a), corresponding to classes more easily understood by the model, like `bad_assign` where the associated mutation replaces a single `=` by a `==` and vice-versa. These labels are well represented inside the training task and precise enough for the model to classify correctly these cases inside real student submissions.

Table 2: Class micro-average scores on the evaluation dataset of the three initilizations.

Score	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

The second group (Figure 3b) gathers labels for which the model obtained poor results. Within this subset, we can find the `miss_parenthesis` and `miss_try` classes we previously highlighted during the first experiment. This observation confirms that the contradictory examples during training could mislead the model when used for real instances. The limited representativeness of these cases during training could also be a cause of bad results obtained by the model. As part of our future work we envision that teachers can modify these mutations to overcome the bad behaviour of this group.



(a) Best performing group

(b) Lowest performing group

(c) Medium performing group

Fig. 3: Average label-wise f1-scores of the models on the evaluation dataset over the three selected seeds.

The last group contains labels with scores between 40% and 80% (Figure 3c). Among these classes, some cases can be identified as suffering from a mutation overfitting phenomenon from the model. Instead of learning the training instances, it started learning the mutation injections. For example, the mutation associated with the label `bad_range` adds a `+1` inside a call to the `range` function. However, some exercises do require to include an upper bound which may require the need of a `+1`. An example is depicted in Listing 1.1 and the model predictions in Table 3. The model wrongly assumes that the `+1` is discriminant to predict this label.

```
def produits(n):
    out = []
    for i in range(n+1):
        for j in range(n+1):
            if i * j == n:
                out.append((i,j))
    return out
```

Listing 1.1: Correct example requiring the use of `+1` inside the call to `range`.

Label	Prediction
`bad_range`	0.952
`miss_parenthesis`	0.129
`failed`	0.087
`correct`	0.067

Table 3: Top-4 labels' scores predicted by the model for the instance shown in Listing 1.1.

Such behaviour is also observed for the class `hardcoded_arg` handling assignment of a function argument as the first instruction of its body. The associated mutation assigns this argument to an integer between 0 and 20. Every case that is too far apart from this mutation is not correctly predicted, as depicted in Listing 1.2 and Table 4.

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

Listing 1.2: Instance number 141 of the evaluation dataset. The argument is hardcoded by the student.

Label	Prediction
`miss_return`	1.000
`print_return`	0.997
`bad_assign`	0.840
`hardcoded_arg`	0.001

Table 4: Top-4 labels' scores predicted by the model for the instance shown in Listing 1.2.

We explored two ways to reduce the mutation overfitting based on training process changes without modifying the mutations we inject:

Adding more mutants. A first solution is to add more instances inside the training task. The model could focus on other aspects and mutations inside a more varied set of source codes. To explore this aspect, we created four additional datasets. Table 5 presents the dataset sizes, the average training time and the average evaluation performance for each dataset. The training time goes up to 7 times the one of the *baseline* for an incremental gain of less than one per cent. The high resource cost of this approach compared to its low gain proves it to be unsuited to counter the mutation overfitting.

Table 5: Results on the evaluation dataset and training time (TT) when adding instances with the same mutations inside the training task.

Metrics	*baseline*	dataset 1	dataset 2	dataset 3	dataset 4
Size	15176	27977	29624	95274	100272
Avg TT	$1h\ 11m\ 56$	$2h\ 16m\ 00$	$2h\ 38m\ 37$	$7h\ 49m\ 13$	$8h\ 34m\ 08$
Avg f1-score	0.707	0.705	0.713	0.715	0.709
baseline gain	+0.00%	−0.28%	+0.85%	+1.11%	+0.28%

Limiting training. An alternative solution to limit mutation overfitting is to train the model for fewer epochs. Looking at the f1-scores on the evaluation dataset (Table 6), only the "5 epochs" configuration managed to outperform slightly the *baseline* (10 epochs). Nonetheless, the marginal gain this configuration obtained does not imply it managed to suppress the mutation overfitting phenomenon we previously highlighted.

Table 6: Class micro-average f1-scores obtained by the models when limiting the number of training epochs.

Model	1 epoch	3 epochs	5 epochs	10 epochs
Run 1	0.400	0.627	0.726	0.706
Run 2	0.429	0.616	0.745	0.692
Run 3	0.398	0.638	0.718	0.723
Mean	0.409	0.627	0.729	0.707

Based on these experiments, modifying the training process is not the optimal solution to overcome the overfitting problem. Integrating human feedback to modify the designed mutations or improve the model's predictions with a continuous learning approach seems to be a more promising strategy to be investigated in more detail as future work.

6 Conclusion

This work provided two main technical contributions: a classification neural network using code embeddings generated on students' submissions and a mutation labelling program. This second process, inspired by DeepBugs [23], can use teacher-defined mutation rules to automatically label instances according to specific injections, hence allowing controlled forms of programming misconception detection. Trained to classify the mutants according to their injections, the model obtained f1-scores above 0.7 on real students' submissions. This result could be considered as an important first step towards automatic feedback generation. The prototype is ready to be deployed on a server and integrated into our autograder to serve classification requests. Each label could be associated with a message to specify what the students should take care of to improve their submissions. The confidence levels predicted for the classes would allow them to gauge the required trust level in the system output. This tool could also be useful for the teaching staff to establish points that should be explained during the tutoring sessions based on statistical information from the model's predictions. All the code produced for this work [28] can be found on GitHub[6].

We identified some limits that could threaten the validity of the results and analyses performed in this work. First, the evaluation dataset contains instances we labelled manually. For some instances one could criticise the validity of the associated classes. Some instances could have been omitted unintentionally by the labeller or could have been considered differently by another annotator.

Secondly, we cannot always ensure that the injections lead to incorrect mutants. For example, removing an except clause that is unused, or removing a return in an unreachable part of the code would not change how the submission behaves. It could, on the other hand, create contradictory examples the model could misclassify, or overfit the mutations predicting thus more false alarms, each at the risk of misleading the student afterwards.

Once our prototype is integrated into our university's autograder, it will become possible to take students' judgements into account to improve the model's decisions. A member of the teaching staff could monitor the predictions and use these to adapt the model, rework specific mutations or add misconceptions currently ignored. Such an approach could help to limit the mutation overfitting, and would put the human in the loop.

Another aspect to explore is how we could interpret the model's predictions. Currently, when predicting a label, the model can highlight the most contributing parts of the input based on the Integrated Gradients computation [30]. This process is only enabled on demand since it requires multiple time-consuming estimation steps and gradient computations. Furthermore, even if it could provide insights about the model decisions, these may not be useful for students each time. Exploring faster or more concrete interpretability techniques could be tackled next.

[6] https://github.com/StevenGuyCap/CESReS-model/tree/main

Acknowledgments. 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.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Akkalyoncu Yilmaz, Z., Wang, S., Yang, W., et al.: 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. pp. 19–24. Association for Computational Linguistics, Hong Kong, China (2019). https://doi.org/10.18653/v1/D19-3004
2. Benetti, G., Roveda, G., Giuffrida, D., Facchinetti, T.: Coderiu: a cloud platform for computer programming e-learning. In: 2019 IEEE 17th International Conference on Industrial Informatics (INDIN). vol. 1, pp. 1126–1132 (Jul 2019). https://doi.org/10.1109/INDIN41052.2019.8972058, iSSN: 2378-363X
3. Boe, B., Hill, C., Len, M., Dreschler, G., Conrad, P., Franklin, D.: Hairball: lint-inspired static analysis of scratch projects. In: Proceeding of the 44th ACM technical symposium on Computer science education. pp. 215–220. SIGCSE '13, Association for Computing Machinery, New York, NY, USA (Mar 2013). https://doi.org/10.1145/2445196.2445265
4. Brito, M., Gonçalves, C.: Codeflex: A Web-based Platform for Competitive Programming. In: 2019 14th Iberian Conference on Information Systems and Technologies (CISTI). pp. 1–6 (Jun 2019). https://doi.org/10.23919/CISTI.2019.8760776, iSSN: 2166-0727
5. Chiodini, L., Moreno Santos, I., Gallidabino, A., Tafliovich, A., Santos, A.L., Hauswirth, M.: A Curated Inventory of Programming Language Misconceptions. In: Proceedings of the 26th ACM Conference on Innovation and Technology in Computer Science Education V. 1. pp. 380–386. ITiCSE '21, Association for Computing Machinery, New York, NY, USA (Jun 2021). https://doi.org/10.1145/3430665.3456343
6. Croft, D., England, M.: 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. pp. 1–4 (Jan 2020). https://doi.org/10.1145/3372356.3372357, arXiv:1911.11085 [cs]
7. Devlin, J., Chang, M.W., Lee, K., Toutanova, K.: 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. pp. 4171–4186. Association for Computational Linguistics, Minneapolis, Minnesota (2019). https://doi.org/10.18653/v1/N19-1423
8. Edwards, S.H., Perez-Quinones, M.A.: Web-CAT: automatically grading programming assignments. In: Proceedings of the 13th annual conference on Innovation and technology in computer science education. p. 328. ITiCSE '08, Association for

Computing Machinery, New York, NY, USA (Jun 2008). https://doi.org/10.1145/1384271.1384371
9. Feng, Z., Guo, D., Tang, D., Duan, N., et al.: CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In: Findings of the Association for Computational Linguistics: EMNLP 2020. pp. 1536–1547. Association for Computational Linguistics, Online (Nov 2020). https://doi.org/10.18653/v1/2020.findings-emnlp.139
10. Gardner, M., Grus, J., Neumann, M., et al.: AllenNLP: A deep semantic natural language processing platform (2017)
11. Guo, D., Ren, S., Lu, S., Feng, Z., Tang, D., Liu, S., et al.: GraphCodeBERT: Pre-training Code Representations with Data Flow (Sep 2020). https://doi.org/10.48550/arXiv.2009.08366
12. Husain, H., Wu, H.H., Gazit, T., Allamanis, M., Brockschmidt, M.: CodeSearchNet challenge: Evaluating the state of semantic code search. arXiv:1909.09436 (2019)
13. Kanade, A., Maniatis, P., Balakrishnan, G., Shi, K.: Learning and evaluating contextual embedding of source code. In: Proceedings of the 37th International Conference on Machine Learning. ICML'20, vol. 119, pp. 5110–5121. JMLR.org (Jul 2020)
14. Karampatsis, R.M., Sutton, C.: SCELMo: Source Code Embeddings from Language Models (Apr 2020). https://doi.org/10.48550/arXiv.2004.13214, arXiv:2004.13214 [cs]
15. Keuning, H., Heeren, B., Jeuring, J.: Strategy-based feedback in a programming tutor. In: Proceedings of the Computer Science Education Research Conference. pp. 43–54. CSERC '14, Association for Computing Machinery, New York, NY, USA (Nov 2014). https://doi.org/10.1145/2691352.2691356
16. Liu, Q., Kusner, M.J., Blunsom, P.: A Survey on Contextual Embeddings (Mar 2020). https://doi.org/10.48550/arXiv.2003.07278
17. Liu, Y., Lapata, M.: 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). pp. 3730–3740. Association for Computational Linguistics, Hong Kong, China (2019). https://doi.org/10.18653/v1/D19-1387
18. Miaschi, A., Dell'Orletta, F.: Contextual and Non-Contextual Word Embeddings: an in-depth Linguistic Investigation. In: Proceedings of the 5th Workshop on Representation Learning for NLP. pp. 110–119. Association for Computational Linguistics (2020). https://doi.org/10.18653/v1/2020.repl4nlp-1.15
19. Mikolov, T., Chen, K., Corrado, G., Dean, J.: Efficient Estimation of Word Representations in Vector Space (Jan 2013). https://doi.org/10.48550/arXiv.1301.3781
20. Paiva, J.C., Leal, J.P., Figueira, A.: Automated Assessment in Computer Science Education: A State-of-the-Art Review. ACM Transactions on Computing Education **22**(3), 34:1–34:40 (Jun 2022). https://doi.org/10.1145/3513140
21. Paszke, A., Gross, S., Massa, F., et al.: PyTorch: an imperative style, high-performance deep learning library. Curran Associates Inc., Red Hook, NY, USA (2019)
22. Peters, M.E., Neumann, M., Zettlemoyer, L., Yih, W.t.: Dissecting Contextual Word Embeddings: Architecture and Representation. In: Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing. pp. 1499–1509. Association for Computational Linguistics, Brussels, Belgium (2018). https://doi.org/10.18653/v1/D18-1179

23. Pradel, M., Sen, K.: DeepBugs: a learning approach to name-based bug detection. Proceedings of the ACM on Programming Languages **2**, 147:1–147:25 (Oct 2018). https://doi.org/10.1145/3276517
24. Rivers, K., Koedinger, K.R.: Data-Driven Hint Generation in Vast Solution Spaces: a Self-Improving Python Programming Tutor. International Journal of Artificial Intelligence in Education **27**(1), 37–64 (Mar 2017). https://doi.org/10.1007/s40593-015-0070-z
25. Shi, Y., Shah, K., Wang, W., Marwan, S., Penmetsa, P., Price, T.: Toward Semi-Automatic Misconception Discovery Using Code Embeddings. In: LAK21: 11th International Learning Analytics and Knowledge Conference. pp. 606–612. Association for Computing Machinery, New York, NY, USA (Apr 2021). https://doi.org/10.1145/3448139.3448205
26. Singh, R., Gulwani, S., Solar-Lezama, A.: Automated feedback generation for introductory programming assignments. In: Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 15–26. PLDI '13, Association for Computing Machinery, New York, NY, USA (Jun 2013). https://doi.org/10.1145/2491956.2462195
27. Staubitz, T., Klement, H., Teusner, R., Renz, J., Meinel, C.: CodeOcean - A versatile platform for practical programming excercises in online environments. In: 2016 IEEE Global Engineering Education Conference (EDUCON). pp. 314–323 (Apr 2016). https://doi.org/10.1109/EDUCON.2016.7474573, iSSN: 2165-9567
28. Steveny, G.: CESReS: Code Embeddings for a Student Recommendation System. Master's thesis, Ecole polytechnique de Louvain, Université catholique de Louvain, Advisors: Mens, Kim; Nijssen, Siegfried. (Apr 2024), http://hdl.handle.net/2078.1/thesis:46100
29. Striewe, M.: An architecture for modular grading and feedback generation for complex exercises. Science of Computer Programming **129**, 35–47 (Nov 2016). https://doi.org/10.1016/j.scico.2016.02.009
30. Sundararajan, M., Taly, A., Yan, Q.: Axiomatic attribution for deep networks. In: Proceedings of the 34th International Conference on Machine Learning - Volume 70. pp. 3319–3328. ICML'17, JMLR.org, Sydney, NSW, Australia (Aug 2017)
31. Talbot, M., Geldreich, K., Sommer, J., Hubwieser, P.: 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. pp. 1–10. WiPSCE '20, Association for Computing Machinery, New York, NY, USA (Oct 2020). https://doi.org/10.1145/3421590.3421604
32. von Wangenheim, C.G., Hauck, J.C.R., et al.: CodeMaster–Automatic Assessment and Grading of App Inventor and Snap! Programs. Informatics in Education **17**(1), 117–150 (2018)
33. Wolf, T., Debut, L., Sanh, V., 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. pp. 38–45. Association for Computational Linguistics, Online (2020). https://doi.org/10.18653/v1/2020.emnlp-demos.6
34. Zhuang, L., Wayne, L., Ya, S., Jun, Z.: A robustly optimized BERT pre-training approach with post-training. In: Proceedings of the 20th Chinese National Conference on Computational Linguistics. pp. 1218–1227. Chinese Information Processing Society of China, Huhhot, China (Aug 2021)