Aalto University

OPEN ACCESS

This is an electronic reprint of the original article.
This reprint may differ from the original in pagination and typographic detail.

Koutcheme, Charles

Training Language Models for Programming Feedback Using Automated Repair Tools

Published in:
Artificial Intelligence in Education

DOI:
10.1007/978-3-031-36272-9_79

Published: 01/01/2023

Document Version
Peer-reviewed accepted author manuscript, also known as Final accepted manuscript or Post-print

Please cite the original version:
Koutcheme, C. (2023). Training Language Models for Programming Feedback Using Automated Repair Tools. In N. Wang, G. Rebolledo-Mendez, N. Matsuda, O. C. Santos, & V. Dimitrova (Eds.), Artificial Intelligence in Education: 24th International Conference, AIED 2023, Tokyo, Japan, July 3–7, 2023, Proceedings (pp. 830–835). (Lecture Notes in Computer Science; Vol. 13916). Springer. https://doi.org/10.1007/978-3-031-36272-9_79

This material is protected by copyright and other intellectual property rights, and duplication or sale of all or part of any of the repository collections is not permitted, except that material may be duplicated by you for your research use or educational purposes in electronic or print form. You must obtain permission for any other use. Electronic or print copies may not be offered, whether for sale or otherwise to anyone who is not an authorised user.

Training Language Models for Programming Feedback Using Automated Repair Tools

Charles Koutcheme[1 [0000−0002−2272−2763]]

Aalto University, Espoo, Finland
charles.koutcheme@aalto.fi

Abstract. In introductory programming courses, automated repair tools (ARTs) are used to provide feedback to students struggling with debugging. Most successful ARTs take advantage of context-specific educational data to construct repairs to students' buggy codes. Recent work in student program repair using large language models (LLMs) has also started to utilize such data. An underexplored area in this field is the use of ARTs in combination with LLMs. In this paper, we propose to transfer the repairing capabilities of existing ARTs to open large language models by finetuning LLMs on ART corrections to buggy codes. We experiment with this approach using three large datasets of Python programs written by novices. Our results suggest that a finetuned LLM provides more reliable and higher-quality repairs than the repair tool used for finetuning the model. This opens venues for further deploying and using educational LLM-based repair techniques.

Keywords: Program Repair · Large Language Models · Computing Education.

1 Introduction

Debugging is particularly challenging and time-consuming for novice programmers [6], and thus educators are eagerly looking for tools and techniques for assisting with the process. One possibility is the use of Automated Repair Tools (ARTs), which are given erroneous codes, which the ARTs then seek to repair. One of the challenges with ARTs, however, is that the proposed fixes can be relatively far from the original buggy code.

Pre-trained large language models (LLMs) - deep neural networks trained on enormous quantities of data - have presented new avenues for program repair. Existing LLMs [10] have shown promising results when finetuned on datasets comprising buggy codes and their corrections. However, employing these models in an educational context requires having access to such annotated datasets. Although many institutions collect and keep track of students' submissions over time, privacy concerns have often limited the possibility of sharing such data with the public (and thus engaging in any annotation process). Consequently, improving the performance of LLM-based program repair techniques has often been limited to seeking effective prompts [11].

In this paper, we introduce a framework to deploy non-propriety/open large language models for program repair in educational contexts. As a workaround for the lack of available annotated datasets, we propose to finetune LLMs on pairs of buggy student code and their repaired versions created by existing ARTs. Figure 1 shows an overview of the framework. In essence, this method facilitates and allows the creation of neural automated repair tools (NARTs) for programming feedback. In this work, we evaluate the proposed framework to understand *to what extent NARTs can address the challenges around repairing student programs?* In particular, we are interested in two real-life deployment scenarios: (1) a scenario in which data collected over one semester in an introductory programming course can be used for providing feedback in another semester and (2) a scenario where only data from another course is available. We thus address the following two research questions concerning NARTs: (RQ1) How do NARTs perform for repairing familiar buggy programs; (RQ2) How do NARTs perform for repairing unfamiliar buggy programs?

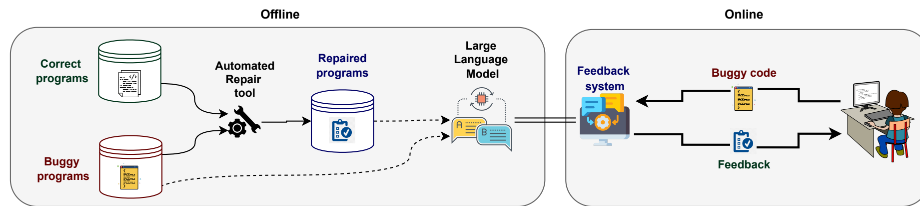

Fig. 1. Overview of the proposed framework. Offline: We use an automated repair tool (ART) to create a dataset of buggy codes and their associated repairs. We use this correction dataset to train an LLM for program repair. Online, a feedback system queries the LLM, providing quality feedback to students.

2 Related work

Automated Repair Tools have been classically built using rule-based systems to construct working solutions from existing data. Most approaches developed for educational contexts use heuristics to match a student's buggy code to a set of correct programs written by other students, or to the instructor's reference solution [4, 5, 8, 9]. An example in this category is the work of Hu et al. [5], whose tool (Refactory) generates a suite of semantically equivalent code by refactoring all available correct solutions to a problem. Then, given an incorrect program, its control flow structure is analysed to find a closely matching working program that can be compared to isolate the buggy components of the program. Previous work has also employed available data with sequence-to-sequence (SEQ2SEQ) models for repairing programs, training a Recurrent Neural Network -based SEQ2SEQ on students' correct solutions [7].

Recently, the large pre-trained language models have gained more attention. For instance, Zhang et al. [11] introduced MMAPR, an automated repair technique for introductory programming assignments. Their approach uses correct solutions, test cases, and assignment descriptions to prompt OpenAI Codex. They evaluate their method on 286 Python programs produced by novices and show that their approach can repair up to 96.5% of the programs, with a smaller edit distance compared to Refactory [5]. Their work is similar in spirit to ours. The differences are that they use Codex (a proprietary system), and their repairing technique uses only a few correct solutions to prompt the system. In contrast, we propose to *finetune* open (i.e. non proprietary) models (such as those available on HuggingFace[1]) using all available data.

3 Methodology

To answer our two research questions, we perform two experiments with three python introductory programming datasets. We use the semantic automated repair tool Refactory [5] to train the language model CodeT5 [10]. CodeT5 is a sequence-to-sequence model pre-trained on a large quantity of bimodal natural and programming language data. As an additional contribution, we release the preprocessed dataset as well as the code to conduct our experiments[2].

Datasets. For our experiments, we use three datasets of student programming solutions written in Python. These datasets were collected from the University of Dublin (DB) [1] over three semesters, from the University of New Caledonia (NC) [3] and the National University of Singapore (SP) [5]. For these datasets, we select the assignments which require writing a single function that takes fixed inputs and produces one output. We further select a subset of assignments from the student datasets to maintain a good ratio of diversity and complexity level.

Experiments. To answer our first research question, we finetune our LLM on the corrections to the solutions of two semesters of data before using it subsequently to find repairs to buggy programs submitted in the following semester. The Dublin dataset contains assignment data from three academic years (2015-2016 to 2017-1018). We use Refactory to find corrections to the buggy programs submitted in the first two academic years (using all the correct solutions of the same two years), forming a training dataset to finetune our LLM. We create a validation set by keeping the submissions of a subset (10%) of students [7]. Finally, we use the finetuned LLM model to repair the buggy solutions submitted in the academic year 2017-2018. We compare our results against those of Refactory when the latter has only access to the original correct programs in the first two academic years. We note that the test set contains exclusively assignments

[1] https://huggingface.co/
[2] https://github.com/KoutchemeCharles/aied2023

seen before during training, while the training set contains data from additional assignments not present in the academic year 2017-2018.

To answer our second research question, we evaluate the model trained on the Dublin dataset to repair the buggy solutions in the Singapore and New Caledonia datasets. We note that the Dublin dataset contains no similar assignments with the Singapore dataset and only two common assignments with New Caledonia dataset. In order to compare our model against Refactory, we provide the ART with the reference solutions for each exercise in the two test datasets.

Technical details. As input to the model, we concatenate the assignment description and the buggy code into a single sequence [10]. For each experiment, we search for the best training hyperparameters using the validation loss. Additionally, for validation and testing, we generate 10 candidate repairs, and we select the best generation parameters on the validation set using the pass@10 metric [2]. The pass@10 metric tells us the probability that *at least one* of the 10 generated repairs passes *all* the unit tests. When testing, out of the ten model outputs, we choose the closest generation to the input buggy program in terms of sequence edit distance. We report our performance on the test sets in terms of success rate, that is, the percentage ratio of programs we manage to repair successfully, and in terms of average sequence edit distance [11].

4 Results and Discussion

Statistics. In the two first academic years at the University of Dublin, students submitted 4269 correct and 3217 buggy programs. Refactory managed to repair 2840 of these buggy programs. We finetuned CodeT5 on the resulting correction dataset aiming to correct 1593, 563, and 933 buggy programs of the test split of the Dublin, New Caledonia and Singapore datasets, respectively.

Results. Table 1 details our results for a subset of the selected assignments. Answering our first research question, when data from the same distribution is available, we observe that the LLM consistently finds higher quality repairs than Refactory (judged in terms of distance) while remaining competitive in terms of the number of repairs found. Answering our second research question, even having never seen most of the assignments before, our model can repair up to 97 % of the buggy programs. We believe the pretraining strategy of LLMs allows them to generalize well to the task of program repair, even when the finetuning data might not be optimal.

Surprisingly, we can observe that even when only given a reference solution (RQ2), Refactory seems to be able to correct almost all buggy codes. Still, these repairs seem to be worse than our model when considering the distance from the original buggy code. When investigating that effect further, we found that, for *both experiments*, Refactory often outputs the reference solution as a correction to buggy codes, even when this reference-based correction does not match the problem-solving intent of the student buggy code. Figure 2 illustrates this issue.

In the example, students had to write a function that removed the duplicates from a list. The original buggy code contains a typo in the second variable "occurrences". The (optimal) repair found by our model correctly identifies this typo but Refactory outputted the reference solution. Although outputting the reference solutions remains a valid functional correction to the buggy code, we argue that these kinds of "repairs" do not align with the spirit of providing valuable feedback for debugging. On the other hand, our model only produces repairs when these can adequately match the student's problem-solving intent. In a nutshell, our model prioritizes quality over quantity.

Table 1. Per assignment experiments results. The left (resp. right) table shows the results for the first (resp. second) experiment. Legend: LLM (Large Language Model), RF (Refactory), SR (success rate), SD (average sequence edit distance), DB (Dublin), NC (New Caledonia), SP (Singapore).

dataset	assignment	RF_SR	LLM_SR	RF_SD	LLM_SD
DB	append2list	1.00	0.96	11.43	12.19
DB	fibonacci_iter	1.00	0.76	56.03	48.65
DB	fibonacci_recur	1.00	1.00	16.79	18.07
DB	index_iter	1.00	0.86	21.36	19.76
DB	index_recur	0.98	1.00	26.11	24.86
DB	maximum	0.96	0.85	26.60	11.36
DB	merge_lists	1.00	0.18	52.45	8.00
DB	minimum	1.00	0.63	24.78	15.00
DB	reverse_iter	1.00	1.00	17.29	17.75
DB	reverse_recur	1.00	0.97	15.79	14.16
DB	search_iter	0.91	0.95	19.65	12.10
DB	search_recur	1.00	1.00	17.41	13.12

dataset	assignment	LLM_SR	RF_SR	LLM_SD	RF_SD
NC	decreasing_list	1.00	0.02	26.63	16.00
NC	is_palindrome	1.00	0.23	43.32	22.86
NC	maximum	0.40	0.19	22.24	15.33
NC	mean	0.77	0.29	33.29	5.29
NC	minimum	0.43	0.20	17.13	10.36
NC	sum	1.00	0.97	11.56	6.91
NC	sum_even_numbers	1.00	0.48	23.14	5.21
NC	sum_n_first_even	1.00	0.21	26.96	2.17
NC	symetrical_list	1.00	0.43	32.70	25.40
SP	remove_extras	1.00	0.01	44.77	22.50
SP	search	0.99	0.37	25.79	5.75
SP	sort_age	0.99	0.17	82.48	4.87
SP	top_k	1.00	0.27	51.30	4.58

Buggy student code

```
1. def remove_extras(lst):
2.     occurrences = ()
3.     new_lst = []
4.     for item in lst:
5.         if item not in occurrences:
6.             occurences += item,
7.             new_list.append(item)
8.     return new_lst
```

LLM repair

```
1. def remove_extras(lst):
2.     occurrences = ()
3.     new_lst = []
4.     for item in lst:
5.         if item not in occurrences:
6.             occurrences += item,
7.             new_list.append(item)
8.     return new_lst
```

ART repair

```
1. def remove_extras(lst):
2.     new_lst = []
3.     for i in lst:
4.         if i not in new_lst:
5.             new_lst.append(i)
6.     return new_lst
```

Fig. 2. Comparing a correction found by a language model against a correction found by an automated repair tool.

5 Conclusion

In this work, we showed that, compared to a classical automated repair tool, using a neural automated repair tool provides: more quality repairs when used in a familiar context (RQ1), and more reliable corrections when used in unfamiliar settings (RQ2).

Limitations and future work. One limitation of our work is that we only used and compared one APR tool with one large language model. Moreover, we did not contrast the performance improvements of this approach against other repairing techniques. On top of addressing these issues, we are currently working on improvements to this framework, namely, using the corrections of multiple repairing tools or other sources to train our models, and experimenting with other types of large language models. We believe that using automated program repair tools as means for training large language models can open new opportunities for applying program repair techniques in education.

References

1. Azcona, D., Smeaton, A.: +5 Million Python & Bash Programming Submissions for 5 Courses & Grades for Computer-Based Exams over 3 academic years. (7 2020). https://doi.org/10.6084/m9.figshare.12610958.v1
2. Chen, M., et al.: Evaluating large language models trained on code (2021). https://doi.org/10.48550/ARXIV.2107.03374
3. Cleuziou, G., Flouvat, F.: Learning student program embeddings using abstract execution traces. In: 14th Int. Conf. on Educ. Data Mining. pp. 252–262 (2021)
4. Gulwani, S., Radiček, I., Zuleger, F.: Automated Clustering and Program Repair for Introductory Programming Assignments (Jun 2018), http://arxiv.org/abs/1603.03165, arXiv:1603.03165 [cs]
5. Hu, Y., Ahmed, U.Z., Mechtaev, S., Leong, B., Roychoudhury, A.: Re-factoring based program repair applied to programming assignments. In: 2019 34th IEEE/ACM Int. Conf. on Automated Software Engineering (ASE). pp. 388–398. IEEE/ACM (2019)
6. McCauley, R., Fitzgerald, S., Lewandowski, G., Murphy, L., Simon, B., Thomas, L., Zander, C.: Debugging: a review of the literature from an educational perspective. Computer Science Education **18**(2), 67–92 (2008). https://doi.org/10.1080/08993400802114581
7. Pu, Y., Narasimhan, K., Solar-Lezama, A., Barzilay, R.: sk_p: a neural program corrector for MOOCs. arXiv:1607.02902 [cs] (Jul 2016), http://arxiv.org/abs/1607.02902, arXiv: 1607.02902
8. 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 (2013). https://doi.org/10.1145/2491956.2462195, https://doi.org/10.1145/2491956.2462195
9. Wang, K., Singh, R., Su, Z.: Data-Driven Feedback Generation for Introductory Programming Exercises. arXiv:1711.07148 [cs] (Nov 2017), http://arxiv.org/abs/1711.07148, arXiv: 1711.07148
10. Wang, Y., Wang, W., Joty, S.R., Hoi, S.C.H.: Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In: EMNLP. pp. 8696–8708. Association for Computational Linguistics (2021)
11. Zhang, J., Cambronero, J., Gulwani, S., Le, V., Piskac, R., Soares, G., Verbruggen, G.: Repairing bugs in python assignments using large language models (2022). https://doi.org/10.48550/ARXIV.2209.14876