Automated Program Repair Using Generative Models for Code Infilling

Charles Koutcheme[1][0000−0002−2272−2763], Sami Sarsa[1][0000−0002−7277−9282], Juho Leinonen[2][0000−0001−6829−9449], Arto Hellas[1][0000−0001−6502−209X], and Paul Denny[2][0000−0002−5150−9806]

[1] Aalto University, Espoo, Finland
`{charles.koutcheme, sami.sarsa, arto.hellas}@aalto.fi`
[2] The University of Auckland, Auckland, New Zealand
`{juho.leinonen, p.denny}@auckland.ac.nz`

Abstract. In educational settings, automated program repair techniques serve as a feedback mechanism to guide students working on their programming assignments. Recent work has investigated using large language models (LLMs) for program repair. In this area, most of the attention has been focused on using proprietary systems accessible through APIs. However, the limited access and control over these systems remain a block to their adoption and usage in education. The present work studies the repairing capabilities of open large language models. In particular, we focus on a recent family of generative models, which, on top of standard left-to-right program synthesis, can also predict missing spans of code at any position in a program. We experiment with one of these models on four programming datasets and show that we can obtain good repair performance even without additional training.

Keywords: Program repair · Large Language Models · Computer Science Education

1 Introduction

Novice programmers will eventually have bugs in their programs which require debugging [14,9]. One possible stream of research for providing support in debugging is the use of large language models (LLMs) [3,4]. Contrary to the use of proprietary LLMs through an API at a cost, we explore *open large language models* that are freely accessible and that can be run locally.

For the present work, we focus on Generative Models for Code Infilling (GMCI) [8] which can be used to fill in missing sections within a given input. Our overarching research goal is to explore the applicability of GMCI for repairing novice programs. We address the following two research questions with respect to GMCI-based program repair techniques: *(RQ1) How do GMCIs perform in fixing programs with a single bug?; (RQ2) with multiple problems and more diverse issues?* In answering the research questions, we also discuss the effect of including metadata information in the prompt and fine-tuning the model on educational data.

2 Background

Automated program repair (APR) research seeks to find ways to automatically fix bugs in programs. Traditional approaches to program repair have utilised test suites to identify defects to fix, followed by the generation of candidate patches that are then validated against the test suite (see e.g. [11,13]). An example in this category is the work of Hu et al. [10], whose tool – Refactory – generates code by refactoring existing solutions to a problem. Then, given an incorrect program, its control flow structure is analysed to find a closely matching solution which is then used to isolate the buggy components of the program.

A variety of machine learning-based methods for program repair exist [5,19], and the recent emergence of LLMs has led to suggestions on using them to localize and fix bugs [15]. In an educational context, Zhang et al. [20] leveraged Codex to fix bugs in Python programming assignments, using correct solutions, test cases, and assignment descriptions for prompting. 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, and with a smaller edit distance compared to other automated program repair approaches [10].

One challenge of the recent proprietary LLMs [18,20] is that the underlying models are opaque and their access is only available through APIs and at a cost. A recent advance that is intuitively applicable for code repair is Generative Models for Code Infilling (GMCI) [2,8] that extend LLMs with infilling capabilities that allow completion within text. This provides an intuitive approach for fixing bugs in place. Such models have been explored and evaluated recently in the context of program repair, where Xia et al [18] showed that LLM-based approaches outperform traditional APR tools.

3 Methodology

In our work, we study GMCIs for program repair in an educational context using the InCoder model [8] available on HuggingFace. The preprocessed data and code for our experiments are released online[3]. In this section, we introduce our data and present the experiments performed to answer our research questions.

Data. We use the QuixBugs dataset [12] and three datasets of student solutions to programming assignments written in Python: (1) Dublin City University data (DB) [1]; (2) University of New Caledonia data (NC) [6]; and (3) National University of Singapore data [10]. We scope our evaluation to assignments that require writing a single function that takes fixed inputs and produces one output and use a subset of assignments to balance diversity and complexity. Additionally, we remove duplicate solutions that could bias evaluation by comparing ASTs.

[3] `https://github.com/KoutchemeCharles/aied2023`

Experiments. To answer our first research question, we use our model for repairing programs on the QuixBugs dataset, where each code contains a single function with a unique bug on a single line. Although this dataset is not strictly a dataset of student programs, many of the bugs present, such as "incorrect assignment operator" and "missing condition", are typical mistakes that students make. We report the number of programs that we can repair, and compare our performance against the work of Prenner and Robbes [15] who used OpenAI Codex with the same data.

To answer our second research question, we evaluate our program repair strategy using InCoder on our student datasets. We report our performance in terms of success rate (i.e. the ratio of the number of programs that we can repair for each assignment in each dataset), and we contrast our results against the Refactory automated program repair tool [10].

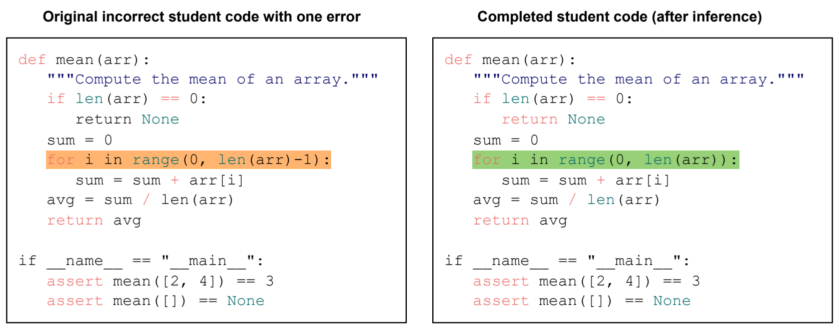
Technical details. To repair a given buggy program, we adapt the multi-line infilling strategy of Fried et al. [8]. Given a function with N lines, our method systematically selects a span of n $(n \leq N)$ consecutive lines and asks the model to complete the code with the missing lines. When completing a program, we generate ten candidate completions using top-p nucleus sampling with $p = 0.95$ and a fixed temperature of 0.6 [4]. A program is considered to be repaired if one of the model-generated solutions passes all automated tests associated with the buggy program. We evaluate up to 50 different spans to find a repair [16]. We refer the reader to the implementation for details of the algorithm.

To allow the model to better understand the buggy program's intended functionality, we add a docstring and we expose the test cases that need to be passed. Figure 1 illustrate our approach. The QuixBugs dataset contains properly formatted docstrings, whereas the Dublin and New Caledonia datasets do not contain docstrings. In these cases, we add single-line docstrings which summarize briefly what the code is supposed to do. For the Singapore dataset, we format the original assignment description as a docstring.

4 Results

Repairing programs with a single bug. Table 1 shows a comparison of InCoder [8] and Codex [15] on the QuixBugs dataset [12]. Our model managed to repair 13 out of the 28 programs. In contrast, Codex performs better with 23 bugs repaired in total.

Repairing student programs. Table 2 breaks down our results per assignment, for a subset of selected assignments, when repairing programs in our student datasets. Overall, although performance varies greatly between different assignments, our results are competitive with the Refactory repair tool (RF), both in terms of the number of repairs found and in terms of distance to the original buggy program.



Original incorrect student code with one error

```
def mean(arr):
   """Compute the mean of an array."""
   if len(arr) == 0:
      return None
   sum = 0
   for i in range(0, len(arr)-1):
      sum = sum + arr[i]
   avg = sum / len(arr)
   return avg

if __name__ == "__main__":
   assert mean([2, 4]) == 3
   assert mean([]) == None
```

Completed student code (after inference)

```
def mean(arr):
   """Compute the mean of an array."""
   if len(arr) == 0:
      return None
   sum = 0
   for i in range(0, len(arr)):
      sum = sum + arr[i]
   avg = sum / len(arr)
   return avg

if __name__ == "__main__":
   assert mean([2, 4]) == 3
   assert mean([]) == None
```

Fig. 1. Prompting our GMCI for repairing student programs. We add a short description of the program functionality as docstring as well as example test cases. We remove part of the buggy code (in orange) and prompt the model to complete the code (in green).

Table 1. Incoder vs Codex on QuixBugs. ✓(resp. ✗) marks bugs which could be successfully (resp. unsuccessfully) repaired. We highlight in gray the results for the assignments which we found were typical in CS1 courses.

	Codex	InCoder
bitcount	✓	✓
bucketsort	✓	✓
find-first-in-sorted	✓	✓
flatten	✓	✗
gcd	✓	✓
get_factors	✓	✓
hanoi	✓	✗
is_valid_parenthes.	✓	✓
kheapsort	✓	✗
knapsack	✓	✗

	Codex	InCoder
kth	✓	✓
lcs_length	✓	✗
levenshtein	✓	✗
lis	✗	✗
long_com_subseq	✓	✗
max_sublist_sum	✓	✓
next_palindrome	✗	✗
next_permutation	✓	✓
pascal	✗	✓

	Codex	InCoder
possible_change	✓	✓
powerset	✓	✗
quicksort	✓	✓
shunting_yard	✗	✓
sieve	✓	✗
sqrt	✓	✗
subsequences	✗	✗
to_base	✓	✗
wrap	✓	✗

Table 2. Comparing program repair performance between our language model (GMCI) and Refactory (RF) in terms of success rate (SR) and average sequence distance (SD).

dataset	assignment_id	RF_SR	GMCI_SR	RF_SD	GMCI_SD
DB	append2list	1.00	0.93	11.43	15.19
DB	fibonacci_iter	1.00	0.94	56.03	33.81
DB	fibonacci_recur	1.00	0.95	16.79	25.41
DB	index_iter	1.00	0.93	21.36	23.83
DB	index_recur	0.98	0.52	26.11	23.66
DB	maximum	0.96	1.00	26.60	33.19
DB	merge_lists	1.00	0.28	52.45	27.51
DB	minimum	1.00	0.97	24.78	25.56
DB	reverse_iter	1.00	0.88	17.29	14.86
DB	reverse_recur	1.00	0.90	15.79	16.92
DB	search_iter	0.91	0.82	19.65	27.67
DB	search_recur	1.00	0.91	17.41	23.26

dataset	assignment_id	RF_SR	GMCI_SR	RF_SD	GMCI_SD
NC	decreasing_list	1.00	0.68	26.63	31.68
NC	is_palindrome	1.00	1.00	43.32	25.97
NC	maximum	0.40	0.96	22.24	26.35
NC	mean	0.77	0.99	33.29	20.61
NC	minimum	0.43	0.96	17.13	22.56
NC	sum	1.00	1.00	11.56	8.53
NC	sum_even_numbers	1.00	0.48	23.14	15.86
NC	sum_n_first_even	1.00	0.96	26.96	17.96
NC	symetrical_list	1.00	0.96	32.70	18.09
SP	remove_extras	1.00	0.42	44.77	32.69
SP	search	0.99	0.81	25.79	24.13
SP	sort_age	0.99	0.67	82.48	40.49
SP	top_k	1.00	0.91	51.30	30.39

5 Discussion and Conclusion

It seems that generative models for code infilling offer good potential for supporting novice programmers and using open large language models straight out of the box performs relatively well for fixing buggy programs. We notice in general that the model's performance depends heavily on the program's complexity, its functionality, and the type of issue(s) encountered in it. Our results are in line with similar work evaluating LLMs for program repair [18,20]. We do not obtain state-of-the-art results, so there is still much room for improvement.

Practical tips. In our preliminary experiments, we experimented with the effect of adding a docstring. Overall, we found that adding a docstring does improve performance. In particular, the more precise the description of the code functionality, the better the results. We also experimented with finetuning the InCoder model on programming data from a previous semester for one of our datasets. Although we notice minor performance improvements on some assignments, we see a degradation in overall performance across all datasets. Instead of fine-tuning, selecting the right generation parameters for the specific dataset can be a better strategy to improve the model performance [8].

Limitations, and future work. One limitation of our work is that we did not compare closed-source state-of-the-art LLMs against InCoder on the student solutions. However, because of the closed nature of these, such a large-scale evaluation is costly. We are also working on developing repair algorithms more adapted for GMCIs, using, for instance, better enumeration strategies [16]. In the spirit of developing more sustainable models, we will also investigate how to create smaller, but perhaps even more efficient, LLMs for program repair. We believe that with a better pre-training strategy, combined with more sophisticated repair algorithms, we can obtain improved performance sufficient for practical use. In the long term, we see great potential in deploying these models in the classroom to address long-standing debugging challenges faced by novices. Indeed, between working on our article in late 2022 and preparing the final submission in May 2023, there has been a growing emphasis on the use and availability of open large language models (e.g. [17,7]).

References

1. Azcona, D., Smeaton, A.: +5 Million Python & Bash Programming Submissions for 5 Courses & Grades for Computer-Based Exams over 3 academic years. (2020). `https://doi.org/10.6084/m9.figshare.12610958.v1`
2. Bavarian, M., Jun, H., Tezak, N., Schulman, J., McLeavey, C., Tworek, J., Chen, M.: Efficient training of language models to fill in the middle (2022). `https://doi.org/10.48550/ARXIV.2207.14255`
3. Bommasani, R., et al.: On the opportunities and risks of foundation models (2021). `https://doi.org/10.48550/ARXIV.2108.07258`

4. Chen, M., et al.: Evaluating large language models trained on code (2021). https://doi.org/10.48550/ARXIV.2107.03374
5. Chen, Z., Kommrusch, S., Tufano, M., Pouchet, L., Poshyvanyk, D., Monperrus, M.: SequenceR: Sequence-to-sequence learning for end-to-end program repair. IEEE Transactions on Software Engineering **47**(09), 1943–1959 (2021). https://doi.org/10.1109/TSE.2019.2940179
6. Cleuziou, G., Flouvat, F.: Learning student program embeddings using abstract execution traces. In: 14th Int. Conf. on Educ. Data Mining. pp. 252–262 (2021)
7. Dey, N., Gosal, G., Khachane, H., Marshall, W., Pathria, R., Tom, M., Hestness, J., et al.: Cerebras-gpt: Open compute-optimal language models trained on the cerebras wafer-scale cluster. arXiv preprint arXiv:2304.03208 (2023)
8. Fried, D., Aghajanyan, A., Lin, J., Wang, S., Wallace, E., Shi, F., Zhong, R., Yih, W.t., Zettlemoyer, L., Lewis, M.: Incoder: A generative model for code infilling and synthesis (2022). https://doi.org/10.48550/ARXIV.2204.05999
9. Hirsch, T., Hofer, B.: A systematic literature review on benchmarks for evaluating debugging approaches. Journal of Systems and Software **192**, 111423 (2022). https://doi.org/https://doi.org/10.1016/j.jss.2022.111423
10. 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) (2019)
11. Le Goues, C., Nguyen, T., Forrest, S., Weimer, W.: Genprog: A generic method for automatic software repair. IEEE Transactions on Software Engineering **38**(1), 54–72 (2012). https://doi.org/10.1109/TSE.2011.104
12. Lin, D., Koppel, J., Chen, A., Solar-Lezama, A.: Quixbugs: A multi-lingual program repair benchmark set based on the quixey challenge. In: Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity. p. 55–56. SPLASH Companion 2017, ACM (2017). https://doi.org/10.1145/3135932.3135941
13. Long, F., Rinard, M.: Automatic patch generation by learning correct code. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. p. 298–312. POPL '16, ACM (2016)
14. 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)
15. Prenner, J.A., Babii, H., Robbes, R.: Can openai's codex fix bugs? an evaluation on quixbugs. In: Proc. of the Third Int. Workshop on Automated Program Repair. pp. 69–75 (2022)
16. Pu, Y., Narasimhan, K., Solar-Lezama, A., Barzilay, R.: Sk_p: A neural program corrector for moocs. In: Companion Proc. of the 2016 ACM SIGPLAN Int. Conf. on Systems, Programming, Languages and Applications: Software for Humanity. p. 39–40. ACM (2016). https://doi.org/10.1145/2984043.2989222
17. Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.A., Lacroix, T., Rozière, B., Goyal, N., Hambro, E., Azhar, F., et al.: Llama: Open and efficient foundation language models. arXiv preprint arXiv:2302.13971 (2023)
18. Xia, C.S., Wei, Y., Zhang, L.: Practical program repair in the era of large pre-trained language models (2022). https://doi.org/10.48550/ARXIV.2210.14179
19. Yasunaga, M., Liang, P.: Graph-based, self-supervised program repair from diagnostic feedback (2020). https://doi.org/10.48550/ARXIV.2005.10636
20. 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