

Investigating the Proficiency of Large Language Models in Formative Feedback Generation for Student Programmers

Smitha S Kumar*

Heriot-Watt University
Dubai, United Arab Emirates
sd102@hw.ac.uk

Manuel Maarek

Heriot-Watt University
Edinburgh, United Kingdom
m.maarek@hw.ac.uk

Michael Adam Lones

Heriot-Watt University
Edinburgh, United Kingdom
m.lones@hw.ac.uk

Hind Zantout

Heriot-Watt University
Dubai, United Arab Emirates
h.zantout@hw.ac.uk

ABSTRACT

Generative AI has considerably altered traditional workplace practice across numerous industries. Ever since the emergence of large language models (LLMs), their potential to generate formative feedback for introductory programming courses has been extensively researched. However, most of these studies have focused on Python. In this work, we examine the bug-fixing and feedback-generation abilities of Code Llama and ChatGPT for Java programming assignments using our new Java benchmark called CodeW Bugs. The results indicate that ChatGPT performs reasonably well, and was able to fix 94.33% programs. By comparison, we observed high variability in the results from Code Llama. We further analyzed the impact of different types of prompts and observed that prompts that included task descriptions and test inputs yielded better results. In most cases, the LLMs precisely localized the bugs and also offered guidance on how to proceed. Nevertheless, we also noticed incorrect responses generated by the LLMs, emphasizing the need to validate responses before disseminating feedback to learners.

CCS CONCEPTS

• **Applied computing** → **Computer-assisted instruction**; • **Computing methodologies** → **Machine translation**; **Natural language generation**.

KEYWORDS

Large language models (LLM), GPT-4, Feedback, Java Programming, Program Repair

ACM Reference Format:

Smitha S Kumar, Michael Adam Lones, Manuel Maarek, and Hind Zantout. 2024. Investigating the Proficiency of Large Language Models in Formative Feedback Generation for Student Programmers. In *2024 International Workshop on Large Language Models for Code (LLM4Code '24)*, April 20, 2024, Lisbon, Portugal. ACM, New York, NY, USA, 6 pages. <https://doi.org/10.1145/3643795.3648380>

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
LLM4Code '24, April 20, 2024, Lisbon, Portugal
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0579-3/24/04.
<https://doi.org/10.1145/3643795.3648380>

1 INTRODUCTION

Large Language models (LLMs) have considerably impacted sectors ranging from healthcare to finance, marketing, software engineering, and education [25]. LLM-based code generators such as OpenAICodex [27], GitHub CoPilot [17] and Code Llama [32] have already garnered significant global attention. The implications of such tools in programming education have created considerable apprehension among academics and researchers in computer science (CS) education [35]. As a result, there has been a notable increase in the number of research papers examining the opportunities and challenges of LLMs in programming education [16, 30].

The recent technological developments in Generative AI suggest that programming will be an essential skill for any computer science graduate. However, programming continues to be a grand challenge for many students in their introductory programming course (commonly called CS1) at university [3]. One reason for this is the absence of immediate personalized feedback in large CS1 classes, which tends to demotivate learners [4]. Even when available, Scheeler [34] suggests that formative feedback is only effective when it is specific, corrective, positive, and immediate. To address this deficit, a variety of methods, including data-driven techniques [31], automatic program repair methodologies [19] and machine learning-based systems [41] have been used to generate formative feedback.

According to Narciss [24], “feedback refers to all post-response information which informs the learner on his/her actual state of learning or performance in order to regulate the further process of learning” and the various types of feedback include knowledge of performance (KP), knowledge of result or response (KR), and knowledge of the correct response (KCR). Elaborated feedback types such as knowledge about task constraints (KTC), knowledge about concepts (KC), knowledge about mistakes (KM), knowledge about how to proceed (KH), and knowledge about meta-cognition (KMC) offer greater guidance to learners. In light of the notable developments in LLMs, researchers are examining viable approaches to integrating these models into programming language instruction [29, 36, 39, 40, 42] to improve programming pedagogy.

Many of the previous research studies indicate that LLMs can generate programming assignments, provide high-quality explanations for code, and even perform defect detection [38]. However, a vast majority of the recent studies have been conducted using

datasets that contain only Python programming assignments. Java¹ is another commonly used programming language in undergraduate studies, and it has different characteristics to Python, including an emphasis on object-oriented programming. To the best of our knowledge, there has not been an evaluation of the bug-fixing and feedback-generation ability of LLMs when applied to Java code. Moreover, there are a limited number of benchmarks such as [5, 11, 23, 28] that are publicly available for Java.

In this work, we primarily investigate the bug-fixing and feedback-generation ability of the popular LLMs Code Llama² and ChatGPT³ using a new benchmark, CodeWBugs⁴.

The following research questions guide this study:

- RQ1. How effective are the LLM systems in generating solutions for CS1 Java programming assignments?
- RQ2. How precisely do the LLMs localize the bug, and what is the nature of the feedback generated?
- RQ3. To what extent does the addition of task descriptions and examples to the prompt improve the LLM's performance?

2 BACKGROUND

Programming is perceived as a complex task that presents significant challenges to CS1 learners [14], with [18] noting that novice programmers find procedural concepts, object-oriented programming concepts, algorithm design concepts, and program design concepts challenging. Research studies [9] have also highlighted that students spend a considerable amount of time resolving errors in Java code and this highlights the need for specific instructor support. These studies highlight the need for clear and concise feedback.

As LLMs have gained popularity in academia, CS education researchers are increasingly investigating the possibility of using LLMs to enhance the student learning experience [1]. The history of Generative AI extends back to the 1950s. However, it was with the introduction of deep learning that generative models gained a notable performance improvement [6]. These models are trained on millions of parameters, and are typically based on the Transformer [37] architecture. Although LLMs were originally designed for natural language processing, they are increasingly being used in various domains such as social science, natural science, engineering, medical applications, and education [8].

With the advent of new LLMs such as GitHub Copilot [17], StarCoder [13] and ChatGPT [26], there is a growing interest in utilizing these tools for code generation and program repair [15]. ChatGPT [26], from OpenAI, is one of the most popular conversational AI systems. Code Llama [32] is a family of open code-specialized large language models based on Llama2 such as Code Llama, Code Llama-Python, and Code Llama-Instruct and available in three sizes (7B, 13B, 34B). These models were evaluated on different programming languages using the MultiPL-E benchmark [7].

Numerous studies have been conducted to evaluate how professional programmers [2] and novice programmers [20] utilize AI code generators. It was observed that novice programmers often

use four different coding approaches to interact with the code generators: AI Single Prompt (prompt the LLMs to generate the full code), AI Step-by-Step (use the LLMs to generate code for subtasks), Hybrid (some part of the code was LLM generated, and the other was manually coded), and Manual (wrote the code before initiating the prompt) [21]. As LLMs are being widely used by novice programmers as a formative feedback generation tool, it is of paramount importance that learners are educated about the implications of over-reliance on these tools. According to Keuning et al. [22], very few existing automated feedback generation tools generate KH feedback. Therefore researchers have investigated the use of LLMs to create effective formative feedback. Although a vast majority of code explanations are correct, these studies also emphasize the need to validate the quality of the generated content [33].

3 METHODOLOGY

This section of the report provides detailed information about the new benchmark and the data analysis approach implemented in this study.

3.1 Data Selection

We created CodeWBugs using the student submissions collected by CodeWrite, a web-based tool designed by the University of Auckland [10] to support novice programmers. We conducted an initial data filtering process to identify a group of incorrect programs and developed JUnit test cases for them. Additionally, incomplete student submissions or multiple submissions with minor modifications in the code were filtered out. A total of 106 programs from 10 programming projects that focus on fundamental Java concepts were considered for this research study (see Table 1). The programs and the corresponding JUnit test cases that we developed are available from the CodeWBugs GitHub repository⁵.

Table 1: Dataset Outline

Project title [ID]	No of programs
productIsEven [PISE]	6
replaceCharacterAtPosition [RCP]	12
countOdds [CO]	7
positionOfValueInArray [PVA]	10
magicNumbers [MN]	14
sumValues [SV]	10
swapEnds [SE]	9
reverse [REV]	6
containerNeeded [CN]	17
weeklyPay [WP]	15
<i>Total</i>	106

The task description [TASK DESCRIPTION] for each of the programming projects are as follows:

productIsEven Complete the productIsEven() method below. This method should calculate the product of the two input values and return true if the product is an even number, and false if the product is an odd number. For example, the

¹<https://uk.indeed.com/career-advice/career-development/coding-languages>

²<https://huggingface.co/codellama/CodeLlama-34b-Instruct-hf>

³<https://chat.openai.com/>

⁴<https://github.com/ssk1216/CodeWBugs>

⁵<https://github.com/ssk1216/CodeWBugs>

product of 10 and 20 is 200, and this is an even number, so the method call `productIsEven(10, 20)` should return `true`.

replaceCharacterAtPosition The `replaceCharacterAtPosition()` method should replace the character at the specified position in the input String with the replacement character provided. For example, `replaceCharacterAtPosition(1, "hello", 'X')` should return the String "hXllo". You can assume that the input String will contain at least one character and that the value of position will be a valid character position in the String. You should make use of the `substring()` method to solve this exercise.

countOdds The `countOdds()` method takes one array of positive integer values as input, and counts how many of those values are odd numbers. For example, if the input array is `1,2,3,4,5` the output will be 3 because 1, 3 and 5 are odd numbers. The array may be empty, in which case the value returned should be 0. Complete the `countOdds()` method below.

positionOfValueInArray The `positionOfValueInArray()` method is passed two parameters - the first is an array ("values") that contains integers, and the second is a number ("find") to search for in the array. If the number is contained in the array, the method should return the index position of the first occurrence of this number. If the number is not contained in the array, the method should return -1. You can assume that the array will contain at least one element. For example, if the input array is: `5,4,3,2,1,4` and the value to search for (i.e. `find`) is 4, then the method should return: 1 which is the index position of the first occurrence of the value 4 in the array.

magicNumbers Two positive integers are said to be magic numbers if they satisfy the following three properties: both numbers are the same length (i.e. consist of the same number of digits) 2) the numbers must be different (i.e. the numbers cannot be the same) 3) the smaller of the two numbers is a divisor/factor (i.e. divides with no remainder) of the larger number For example, the following pairs of numbers are "magic": 900, 450 3000, 1000 90, 30. However, the following pairs of numbers are not "magic " because they violate at least one of the three properties listed above: 100, 100 (these two numbers are the same) 1000, 100 (the number of digits in each number is different) 200, 101 (the smaller number is not a factor of the larger number). Complete the `magicNumbers()` method which takes two input numbers and returns true if they are magic numbers and false otherwise. You can assume that both input values will be positive.

sumValues The `sumValues()` methods takes two inputs: an array of integers, and a boolean called `positivesOnly`. The method should calculate the sum of the values in the array, however if `positivesOnly` is true, then any negative numbers in the array should be ignored and not contribute to the sum. For example, if the input array is `-1,2,-3,4,-5,6`, then calling `sumValues()` with `positivesOnly` being true should give 12 because only the positive values (2,4,6) are summed, whereas if `positivesOnly` is false the output should be: 3 ,because all of the values in the array are summed. Note: the array may be empty in which case the method should return 0.

swapEnds Complete the method called `swapEnds()` which takes one String as input. It must return a new String, which is the same as the input String except that the first and last characters of the String are swapped. For example, if the input String is "Hello" , the output String must be "oellH".

reverse Mary creates an array of integers, however she realises that the values have been stored in the array in the opposite order to what she needs. She therefore would like to define a method to reverse the order of the values in the array. Her method, called `reverse()`, will be passed an array as input and will return a new array (note: the original input array is not changed) which contains the same values as the original input array but in reverse order. For example if the input array is: `1,2,3,4,5`, the new array returned by the method will be `5,4,3,2,1`.

containersNeeded A factory requires a program to calculate the number of containers needed to store a given number of items. Each container can fit up to 10 items. In this exercise, you need to complete the function `containersNeeded()` which is passed the number of items to store as an input parameter. The function should calculate and return the number of containers sufficient to store that many items. For example, if there are 10 items to store, then only 1 container is needed. However, if there are 11 items to store, then 2 containers are needed (one container will be full, and the other will have just one item).

weeklyPay In this exercise, the pay that an employee earns each week depends on their age, the number of hours worked during normal business hours, and the number of overtime hours worked. The base pay rate for all workers is \$15 per hour. On top of the base rate, each worker over the age of 20 earns an extra \$1 per hour for every year their age exceeds 20. So, for example, a worker who is 25 years old, will receive a base pay rate of $15 + 5 = \$20$ per hour. However, this additional, age-based bonus is only valid up until the age of 40. So, for example, a 40 and a 45-year-old will earn the same base rate. Finally, any overtime hours are paid at twice the base rate. Complete the `weeklyPay()` method below which calculates the amount paid to an employee in one week, based on the number of normal and overtime hours worked, as well as the age. You should work in whole numbers (integers) only. if $(age \leq 20)$ $extra = 0$ elif $(age > 20 \text{ and } age \leq 40)$: $extra = age - 20$ $(age > 40)$: $extra = 20$ $baseWithAge = 15 + extra$ $result = (normalHours * baseWithAge) + (overtimeHours * baseWithAge * 2)$.

3.2 Data Analysis

For RQ1, the basic prompt (*prompt1*) is designed as follows:

Provide a fix for the buggy version of Java code
[SOURCE CODE]

where [SOURCE CODE] represents the student submission.

Prompt1 is initially inputted to each of the LLMs without any supplementary information related to the bug.

For example, *prompt1* for a PISE program is as follows:

Provide a fix for the buggy version of Java code
"boolean productIsEven(int a, int b){

```

        int product = 2 *(a * b);
        boolean evenOr = product == 1;
        return evenOr;
    }

```

The generated response was evaluated against the JUnit test cases in the CodeW Bugs benchmark. For responses that led to a JUnit test failure, a modified prompt (*prompt2*) that includes a task description along with the student submission was provided as input.

The format of *prompt2* is as follows:

```

[TASK DESCRIPTION]
Provide a fix for the buggy version of Java code
[SOURCE CODE]

```

The task descriptions and the JUnit test cases are available on the GitHub page for CodeW Bugs. The generated response was again validated for correctness using JUnit tests. The prompt and the associated response generated by the LLMs were recorded in each iteration.

The basic prompt for RQ2 (*prompt3*) is:

```

What is the error in the Java code?
[SOURCE CODE]

```

where the source code is the student submission that is buggy. Additional information about the bug was not provided to the LLM along with the prompt. For certain failed test results, additional dialogue with the LLMs was initiated to generate accurate fixes and feedback. The generated responses were examined for their accuracy in bug localization, code correction tips, and any additional information to improve the code quality.

4 RESULTS

4.1 Bug Fixing Ability: RQ1

To answer RQ1, we examined the bug-fixing ability of the LLMs using CodeW Bugs. Of the 106 programs, chatGPT successfully fixed 94.33% of programs using *prompt1* and *prompt2* combined, while Code Llama fixed 66.03% of programs. Table 2 provides an overview of the number of programs successfully fixed by the LLMs using *prompt1* and *prompt2* for each of the programming projects.

Code Llama initially fixed 54 programs with *prompt1*, and for those programs that failed to generate a correct code, *prompt2* was used as input, resulting in an increase in the count to 70 programs. Similarly, the number of programs fixed by ChatGPT increased from 91 to 100. Code Llama and ChatGPT were able to generate an accurate fix for most of the programs in the *PISE* category with the initial version of the prompt (*prompt1*). Both the LLMs were also able to achieve high repair rates for programs that involved array-based manipulation, such as *CO* and *REV* with *prompt1*. Although both LLMs performed well on the string handling tasks such as *RCP* and *SE*, ChatGPT achieved a higher repair rate than Code Llama.

Code Llama failed to generate an accurate fix for most of the programs in the *MN* (1/14) and *WP* categories (5/15). These two categories of programs use specific formulae that the LLMs could not correctly identify using the initial prompt. ChatGPT also struggled to generate an accurate fix for the *MN* category of programs using *prompt1*; however, the repair rate was high when *prompt2* was used. Thus we can observe that there is high variability in the repair rate across the projects. In certain cases, Code Llama generated partial

Table 2: Code Llama-34b-Instruct-hf vs. ChatGPT

CodeW Bugs ID	#programs	Code Llama		ChatGPT	
		<i>prompt1</i>	<i>prompt2</i>	<i>prompt1</i>	<i>prompt2</i>
PISE	6	5	1	6	
RCP	12	8	3	12	
CO	7	7		7	
MN	14		1		14
SV	10	7		10	
SE	9	4	2	9	
REV	6	6		6	
CN	17	9	4	16	1
WP	15	1	4	7	8
PVA	10	7	1	10	

output or output that included the `main()` method; however, with additional dialogue, these were fixed.

4.2 Types of feedback: RQ2

According to Narciss [24], feedback types such as knowledge about how to proceed (KH) and knowledge about mistakes (KM) are more beneficial to novice programmers than knowledge of the correct response (KCR) type feedback. We conducted a detailed analysis of the generated responses for two programming exercises (*PISE* and *RCP*) using *prompt3*. These two projects were selected because of the high repair rate achieved by the LLMs in the initial experiment. 60 responses were analyzed to identify the number of programs with precisely localized bugs. The generated feedback was analyzed and classified according to the terminology proposed by Narciss [24]. Table 3 provides a summary of the findings.

Table 3: Accuracy and type of feedback

ID	Code Llama	GPT4	Feedback type
PISE	5	6	KCR, KM, KH
RCP	7	11	KCR, KM, KH

Of the 6 programs in *PISE*, Code Llama was able to localize the bug in three *PISE* programs accurately. Including the task description in the subsequent interactions with Code Llama resulted in a higher number of bugs (total of 5) being localized correctly. ChatGPT, on the other hand, was able to localize the bug accurately for all 6 programs. For example, for an incorrect code sample in the *PISE* project:

```

boolean productIsEven(int a, int b){
    int product = 2 *(a * b);
    boolean evenOr = product == 1;
    return evenOr;
}

```

the corresponding ChatGPT-generated response was as follows: "This line calculates the remainder when 2 is divided by the product of a and b. However, this is not the correct way to check for evenness. To check if a number is even, you should check if the number divided by 2 has a remainder of 0."

The generated response includes all three components: it localizes the bug (KM), generates a correct response (KCR) and also guides the learner (KH) in the right direction. However, in some instances, the responses generated by the LLMs are incorrect.

For the same buggy code above, the response generated by Code Llama was: *"The error in the Java code is that the modulo operator (%) has higher precedence than multiplication. Therefore, 2 % (a * b) will first compute the result of a * b, and then apply the modulo operation on it with 2. This means that if a and b are both odd numbers, the result of product will always be 0, regardless of whether their product is even or not"*. Additionally, Code Llama provided stylistic suggestions, such as identifying unused variables, applying naming conventions, and improving the overall code structure. For the second project, RCP, out of the 12 programs, ChatGPT could successfully localize the bug for 11 programs using the initial prompt while Code Llama managed to do so for 7 programs.

For example, for an incorrect code sample in the RCP project:

```
String replaceCharacterAtPosition(int position,
String word, char replacement){
    String replacedLetter =
    word.substring(0,position) + replacement +
    word.substring(position+1,
    word.length() - 1);
    return replacedLetter;
}
```

The response generated by ChatGPT was: *"The error in the provided Java code lies in the way the string slicing is performed for replacing a character at a given position. Specifically, the issue is with the expression word.substring(0, position-1). In Java, the substring method is zero-indexed, meaning that indexes start at 0. Therefore, if you want to replace the character at position, you should not subtract 1 from position in the first part of the substring. The correct way to do it would be word.substring(0, position). This ensures that the substring includes all characters up to, but not including, the character at position. Here's the corrected version of your method:*

```
String replaceCharacterAtPosition(int position,
String word, char replacement){
    String newWord =
    word.substring(0, position)
    + replacement +
    word.substring(position + 1);
    return newWord;
}"
```

The generated response includes all three components: KCR, KM and KH. These results cannot be generalized, as there are multiple instances where the feedback was either incorrect or the generated code used a different approach than the original code. In the majority of the instances with Code Llama, the response generated for the initial prompt was incorrect, and initiating a dialogue with extra information about the fault resulted in better responses.

As LLMs are probabilistic [12], there might be a lot of variability in the results for the same set of prompts. This could be mitigated by running the same prompt repeatedly. Moreover, LLMs in most cases generated the KCR feedback, and this could negatively impact the learning experience. Despite the challenges, LLMs seem to

have the potential to address the extensively researched problem of generating effective formative feedback in CS1 education.

4.3 Merits of including additional information with the prompt: RQ3

We observed that there was a substantial increase in the bug-fixing ability of Code Llama when advanced prompts were used in comparison to ChatGPT. For example, Code Llama successfully generated the correct patches for an additional 16 programs that failed when *prompt1* was used. ChatGPT was able to fix an additional nine programs using *prompt2*. We also observed that programs in the MN and WP categories used certain task-specific formulae. The LLM's performance was notably deficient for these categories using *prompt1*, as shown in Table 4.

Table 4: Prompt1 based LLM evaluation results

Project ID	No of programs	Code Llama	ChatGPT
MN	14	0	0
WP	15	1	7

The inclusion of [TASK DESCRIPTION] in the initial prompt boosted the performance of Code Llama and ChatGPT significantly, as shown in Table 2. This highlights the need to craft effective prompts, which might be a challenging task for novice programmers. Therefore, students must be taught the art of constructing effective prompts to fully utilize the power of LLMs.

5 CONCLUSION

In this paper, we examined the effectiveness of two state-of-the-art LLMs, Code Llama (code Llama-34b-Instruct-hf) and ChatGPT (GPT4), in generating precise feedback for CS1 Java programming assignments. This study could be extended to include other code LLMs. Additionally, we introduced a new benchmark CodeW Bugs that could be used for future research purposes. The findings of our study emphasize the need for a comprehensive validation of the generated feedback because of the high variability in the response. Further studies are needed to overcome these challenges and to design effective systems using LLMs in programming education.

REFERENCES

- [1] Rishabh Balse, Bharath Valaboju, Shreya Singhal, Jayakrishnan Madathil Warriem, and Prajish Prasad. 2023. Investigating the Potential of GPT-3 in Providing Feedback for Programming Assessments. In *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1* (, Turku, Finland,) (ITiCSE 2023). Association for Computing Machinery, New York, NY, USA, 292–298. <https://doi.org/10.1145/3587102.3588852>
- [2] Shradha Barke, Michael B. James, and Nadia Polikarpova. 2022. Grounded Copilot: How Programmers Interact with Code-Generating Models. arXiv:2206.15000 [cs.HC]
- [3] Brett A. Becker, Paul Denny, James Finnie-Ansley, Andrew Luxton-Reilly, James Prather, and Eddie Antonio Santos. 2023. Programming Is Hard - Or at Least It Used to Be: Educational Opportunities and Challenges of AI Code Generation. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1* (Toronto ON, Canada) (SIGCSE 2023). Association for Computing Machinery, New York, NY, USA, 500–506. <https://doi.org/10.1145/3545945.3569759>
- [4] Brett A Becker, Paul Denny, Raymond Pettit, Durell Bouchard, Dennis J Bouvier, Brian Harrington, Amir Kamil, Amey Karkare, Chris McDonald, Peter-Michael Osera, et al. 2019. Compiler error messages considered unhelpful: The landscape of text-based programming error message research. *Proceedings of the working group reports on innovation and technology in computer science education* (2019), 177–210. <https://doi.org/10.1145/3344429.3372508>

- [5] Neil Christopher Charles Brown, Michael Kölling, Davin McCall, and Ian Utting. 2014. Blackbox: A large scale repository of novice programmers' activity. In *Proceedings of the 45th ACM technical symposium on Computer science education*. 223–228. <https://doi.org/10.1145/2538862.2538924>
- [6] Yihan Cao, Siyu Li, Yixin Liu, Zhiling Yan, Yutong Dai, Philip S. Yu, and Lichao Sun. 2023. A Comprehensive Survey of AI-Generated Content (AIGC): A History of Generative AI from GAN to ChatGPT. arXiv:2303.04226 [cs.AI]
- [7] Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q Feldman, Arjun Guha, Michael Greenberg, and Abhinav Jangda. 2022. MultiPL-E: A Scalable and Extensible Approach to Benchmarking Neural Code Generation. arXiv:2208.08227 [cs.LG]
- [8] Yupeng Chang, Xu Wang, Jindong Wang, Yuan Wu, Linyi Yang, Kaijie Zhu, Hao Chen, Xiaoyuan Yi, Cunxiang Wang, Yidong Wang, Wei Ye, Yue Zhang, Yi Chang, Philip S. Yu, Qiang Yang, and Xing Xie. 2023. A Survey on Evaluation of Large Language Models. arXiv:2307.03109 [cs.CL]
- [9] Paul Denny, Andrew Luxton-Reilly, and Ewan Tempero. 2012. All syntax errors are not equal. In *Proceedings of the 17th ACM annual conference on Innovation and Technology in Computer Science Education*. 75–80. <https://doi.org/10.1145/2325296.2325318>
- [10] Paul Denny, Andrew Luxton-Reilly, Ewan Tempero, and Jacob Hendrickx. 2011. CodeWrite: Supporting Student-Driven Practice of Java. In *Proceedings of the 42nd ACM Technical Symposium on Computer Science Education* (Dallas, TX, USA) (SIGCSE '11). Association for Computing Machinery, New York, NY, USA, 471–476. <https://doi.org/10.1145/1953163.1953299>
- [11] Thomas Durieux and Martin Monperrus. 2016. IntroClassJava: A benchmark of 297 small and buggy Java programs. <https://hal.archives-ouvertes.fr/hal-01272126>
- [12] Mark Chen et al. 2021. Evaluating Large Language Models Trained on Code. arXiv:2107.03374 [cs.LG]
- [13] Raymond Li et al. 2023. StarCoder: may the source be with you! arXiv:2305.06161 [cs.CL]
- [14] Andrew Ettles, Andrew Luxton-Reilly, and Paul Denny. 2018. Common logic errors made by novice programmers. In *Proceedings of the 20th Australasian Computing Education Conference*. 83–89. <https://doi.org/10.1145/3160489.3160493>
- [15] Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M. Zhang. 2023. Large Language Models for Software Engineering: Survey and Open Problems. arXiv:2310.03533 [cs.SE]
- [16] James Finnie-Ansley, Paul Denny, Brett A Becker, Andrew Luxton-Reilly, and James Prather. 2022. The robots are coming: Exploring the implications of openai codex on introductory programming. In *Proceedings of the 24th Australasian Computing Education Conference*. 10–19. <https://doi.org/10.1145/3511861.3511863>
- [17] Nat Friedman. 2021. *Introducing GitHub Copilot: your AI pair programmer*. <https://github.blog/2021-06-29-introducing-github-copilot-ai-pair-programmer>
- [18] Ken Goldman, Paul Gross, Cinda Heeren, Geoffrey Herman, Lisa Kaczmarczyk, Michael C Loui, and Craig Zilles. 2008. Identifying important and difficult concepts in introductory computing courses using a delphi process. In *Proceedings of the 39th SIGCSE technical symposium on Computer science education*. 256–260. <https://doi.org/10.1145/1352322.1352226>
- [19] Sumit Gulwani, Ivan Radiček, and Florian Zuleger. 2018. Automated clustering and program repair for introductory programming assignments. *ACM SIGPLAN Notices* 53, 4 (2018), 465–480. <https://doi.org/10.1145/3192366.3192387>
- [20] Majeed Kazemitabaar, Justin Chow, Carl Ka To Ma, Barbara J. Ericson, David Weintrop, and Tovi Grossman. 2023. Studying the Effect of AI Code Generators on Supporting Novice Learners in Introductory Programming. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems* (Hamburg, Germany) (CHI '23). Association for Computing Machinery, New York, NY, USA, Article 455, 23 pages. <https://doi.org/10.1145/3544548.3580919>
- [21] Majeed Kazemitabaar, Xinying Hou, Austin Henley, Barbara J. Ericson, David Weintrop, and Tovi Grossman. 2023. How Novices Use LLM-Based Code Generators to Solve CS1 Coding Tasks in a Self-Paced Learning Environment. arXiv:2309.14049 [cs.HC]
- [22] Hieke Keuning, Johan Jeuring, and Bastiaan Heeren. 2018. A systematic literature review of automated feedback generation for programming exercises. *ACM Transactions on Computing Education (TOCE)* 19, 1 (2018), 1–43. <https://doi.org/10.1145/3231711>
- [23] Derrick Lin, James Koppel, Angela Chen, and Armando Solar-Lezama. 2017. 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*. 55–56. <https://doi.org/10.1145/3135932.3135941>
- [24] Susanne Narciss. 2008. Feedback strategies for interactive learning tasks. In *Handbook of research on educational communications and technology*. Routledge, 125–143.
- [25] Keng-Boon Ooi et al. 2023. The Potential of Generative Artificial Intelligence Across Disciplines: Perspectives and Future Directions. *Journal of Computer Information Systems* 0, 0 (2023), 1–32. <https://doi.org/10.1080/08874417.2023.2261010>
- [26] OpenAI. 2022. *Introducing ChatGPT*. <https://openai.com/blog/chatgpt> Accessed on August 21, 2023.
- [27] OpenAI. 2022. *OpenAI Codex*. <https://openai.com/blog/openai-codex> Accessed on August 21, 2023.
- [28] José Carlos Paiva, José Paulo Leal, and Álvaro Figueira. 2023. PROGPedia: Collection of source-code submitted to introductory programming assignments. *Data in Brief* 46 (2023), 108887. <https://doi.org/10.1016/j.dib.2023.108887>
- [29] Tung Phung, José Cambroner, Sumit Gulwani, Tobias Kohn, Rupak Majumdar, Adish Singla, and Gustavo Soares. 2023. Generating High-Precision Feedback for Programming Syntax Errors using Large Language Models. (2023). <http://arxiv.org/abs/2302.04662>
- [30] James Prather, Paul Denny, Juho Leinonen, Brett A. Becker, Ibrahim Albluwi, Michelle Craig, Hieke Keuning, Natalie Kiesler, Tobias Kohn, Andrew Luxton-Reilly, Stephen MacNeil, Andrew Peterson, Raymond Pettit, Brent N. Reeves, and Jaromir Savelka. 2023. The Robots are Here: Navigating the Generative AI Revolution in Computing Education. arXiv:2310.00658 [cs.CY]
- [31] Thomas W Price, Yihuan Dong, Rui Zhi, Benjamin Paaßen, Nicholas Lytle, Veronica Cateté, and Tiffany Barnes. 2019. A comparison of the quality of data-driven programming hint generation algorithms. *International Journal of Artificial Intelligence in Education* 29 (2019), 368–395. <https://doi.org/10.1007/s40593-019-00177-z>
- [32] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xi-aoping Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnæve. 2023. Code Llama: Open Foundation Models for Code. arXiv:2308.12950 [cs.CL]
- [33] Sami Sarsa, Paul Denny, Arto Hellas, and Juho Leinonen. 2022. Automatic Generation of Programming Exercises and Code Explanations Using Large Language Models. In *Proceedings of the 2022 ACM Conference on International Computing Education Research - Volume 1* (Lugano and Virtual Event, Switzerland) (ICER '22). Association for Computing Machinery, New York, NY, USA, 27–43. <https://doi.org/10.1145/3501385.3543957>
- [34] Mary Catherine Scheeler, Kathy L Ruhl, and James K McAfee. 2004. Providing performance feedback to teachers: A review. *Teacher education and special education* 27, 4 (2004), 396–407. <https://doi.org/10.1177/088840640402700407>
- [35] Adele Smolansky, Andrew Cram, Corina Radulescu, Sandris Zeivots, Elaine Huber, and Rene F. Kizilcec. 2023. Educator and Student Perspectives on the Impact of Generative AI on Assessments in Higher Education. In *Proceedings of the Tenth ACM Conference on Learning @ Scale* (Copenhagen, Denmark) (L@S '23). Association for Computing Machinery, New York, NY, USA, 378–382. <https://doi.org/10.1145/3573051.3596191>
- [36] Dominik Sobania, Martin Briesch, Carol Hanna, and Justyna Petke. 2023. An analysis of the automatic bug fixing performance of chatgpt. In *2023 IEEE/ACM International Workshop on Automated Program Repair (APR)*. <https://doi.org/10.1109/APR59189.2023.00012>
- [37] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2023. Attention Is All You Need. arXiv:1706.03762 [cs.CL]
- [38] Michel Wermelinger. 2023. Using GitHub Copilot to solve simple programming problems. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*. 172–178. <https://doi.org/10.1145/3545945.3569830>
- [39] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated program repair in the era of large pre-trained language models. In *Proceedings of the 45th International Conference on Software Engineering (ICSE 2023)*. Association for Computing Machinery. <https://doi.org/10.1109/ICSE48619.2023.00129>
- [40] Chunqiu Steven Xia and Lingming Zhang. 2023. Conversational automated program repair. (2023). <http://arxiv.org/abs/2301.13246>
- [41] Jooyong Yi, Umair Z Ahmed, Amey Karkare, Shin Hwei Tan, and Abhik Roychoudhury. 2017. A feasibility study of using automated program repair for introductory programming assignments. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 740–751. <https://doi.org/10.1145/3106237.3106262>
- [42] Jialu Zhang, José Cambroner, Sumit Gulwani, Vu Le, Ruzica Piskac, Gustavo Soares, and Gust Verbruggen. 2022. Repairing bugs in python assignments using large language models. (2022). <https://doi.org/abs/2209.14876>