

LLM Tutors for Computer Science Courses

Isaac Pan

Abstract:

Chatbots can provide quick feedback and also answer follow-up questions for students. This study aims to explore the use of chatbots for courses by taking advantage of the pre-defined course assignments to give the chatbot a sample solution to reduce errors. The study focuses on the ability of chatbots to provide high-level plans, pseudocode, and debug students' code. The results showed that the chatbot excelled at providing high-level plans and pseudocode that was often very similar to the sample solution. However, the chatbot also encountered some limitations as it struggled to accurately find errors when debugging and sometimes would incorrectly flag a portion of code that was correct. Overall, chatbots have potential in computer science course education as an assistant, but still require more improvement until it is reliable.

1. Introduction:

The recent developments of Large Language Models (LLMs) such as ChatGPT has opened new possibilities for helping students in academic settings. In introductory Computer Science (CS) courses, where students often have programming assignments, the availability of LLMs creates the possibility of having a digital helper capable of offering immediate support without the delays associated with traditional methods.

This paper explores the potential of LLMs in education, specifically for well-defined material to reduce hallucinations, which is when LLMs provide false information. I used OpenAI's Assistants API to create a chatbot that could respond to users. I used OpenAI's retrieval augmented generation (RAG) tool with the information on the course to allow the model to be more accurate and specially geared for the AP Computer Science course. In addition to adding the lessons and assignments from each course, sample solutions are also added for the logic-based assignments for the chatbot to reference to reduce mistakes.

2. Related Works:

The integration of chatbot interfaces for LLMs represents a significant advancement in providing immediate and interactive support to students on homework assignments, particularly in the realm of computer science education.

Generative models, including ChatGPT, OpenAI Codex, DeepMind AlphaCode, Amazon CodeWhisperer, and GitHub Copilot, have emerged as promising tools for enriching the learning experience of students across several subjects in computing education. Educators have increasingly leveraged these models to accelerate content generation and enhance the acquisition of relevant skills in introductory CS courses. Previous research has examined the utilization of LLMs in generating code explanations, providing immediate feedback, and improving the clarity of error messages from the compiler [1-5].

However, while the availability of LLMs offers students immediate access to assistance on programming assignments, their naive use may not align with pedagogical objectives. Pedagogical objectives describe what a student should be able to do after completing an educational activity such as homework. However, students may use LLMs to generate code and therefore learn nothing. A recent workshop paper¹, reports on the development and preliminary

deployment of a GPT-4-based interactive homework assistant for students in a large introductory computer science course. The paper introduces an assistant equipped with a "Get Help" button within a popular code editor and a "get feedback" feature within a command-line autograder. The assistant aims to provide guidance that supports pedagogical goals while avoiding the direct provision of solutions to students. The study highlights the assistant's ability to identify students' conceptual struggles and offer suggestions, plans, and template code in pedagogically appropriate ways [5].

This paper aims to build upon previous research on using LLMs in computer science courses by including lessons, homework assignments, and sample solutions to reduce LLM. The goal of the research is to use well-defined course material to give the LLM a reference which will aid in improving its accuracy. The LLM will also be able to look at sample solutions to explain logic or high-level plans without giving the answer, thereby being a helpful resource for students that is accessible at any time.

3. Methodology:

The methodology for developing and evaluating the Java homework assistant involved a systematic approach centered on the collection, curation, and integration of educational resources from Cupertino High School's AP Computer Science curriculum. The initial phase focused on gathering comprehensive course materials, including lessons, assignments, and sample solutions, which were stored in separate files (`lessons.txt`, `assignments.txt`, `sampleSolutions.txt`). These resources formed the foundational knowledge base for training the chatbot.

The development of the chatbot was facilitated using the Assistants API provided by OpenAI. This API served as the framework for training the model using the collected course material and sample solutions. The objective was to equip the chatbot with the capability to comprehend and effectively respond to user queries related to Java programming concepts covered in the curriculum.

To enhance the accuracy and relevance of the chatbot's responses, a retrieval tool offered by the Assistants API was integrated into the system. This tool allowed the chatbot to extract specific information from the course material based on user inquiries, thereby providing tailored assistance aligned with the curriculum content. OpenAI's code interpreter tool was also used to assist the chatbot's performance by allowing the chatbot to write and run Python code in a sandboxed execution environment.

During interactions with students, the chatbot followed a structured approach. It began by identifying the specific lab assignment the student was working on, referencing `assignments.txt` for details. If necessary, the chatbot prompted the student to specify the lab. Using the lab information, the chatbot identified the corresponding lesson(s) from `lessons.txt` to provide contextual assistance. If students provided code, the chatbot analyzed it for errors or conceptual misunderstandings, guiding them toward understanding the correct approach without directly providing solutions. For problem-solving, the chatbot referred to `sampleSolutions.txt` and explained concepts using the logic from sample solutions, ensuring transparency about its source.

The effectiveness of the chatbot was evaluated based on two key criteria: accuracy and alignment with pedagogical goals. Accuracy referred to the chatbot's ability to provide correct and helpful responses, while alignment with pedagogical goals assessed the extent to which the

chatbot's interactions supported student learning and problem-solving skills within the AP computer science curriculum.

4. Results:

Based on their accuracy and student feedback, I assessed the usefulness of LLMs in computer science education. The chatbot assistants were given a total of 8 lab assignments and 3 questions per lab. Each lab got a request for a high-level plan, pseudocode, and debugging. The accuracy and quality of their answers were evaluated on a scale from 1 to 5 with 5 being the most accurate or helpful answer possible and 1 being the least accurate or helpful answer. The output file can be found here <https://github.com/Sorcen7/Java-tutor-project>.

For the high-level plan requests, the responses from the chatbot were consistently accurate, mostly being fully accurate while occasionally outputting minor mistakes. The average score for the chatbot's high-level plans was 4.875. On average, the high-level plan also tended to be very helpful as it provided a good starting structure for the student, the steps were logically connected, and the explanation also discussed finer details in the implementation.

The responses to the pseudocode requests were also very accurate since many of the responses directly reflected the sample solutions. The average accuracy score of the pseudocode responses is 5. Most pseudocode responses were in the right format of plain text mirroring the logical flow of a program and also included an explanation, either line-by-line or at the end. However, some responses were not as helpful, as one was in the same format as a high-level plan and the other was code instead of pseudocode. The average rating for how helpful the pseudocode responses are is 4.125.

The responses to the debugging prompts were almost as accurate as the responses to the other questions. A significant portion of the responses included points that were not a problem, so it was not fully accurate, but captured the important errors as well for the most part. The average accuracy score for debugging responses was 4.375. The debugging responses also properly explained the issue rather than just returning a corrected version, which would help students understand what was wrong and how to fix it. The average helpfulness score of the debugging responses was 4.25

Below is an example of a LLM response and an associated evaluation:

User: Provide a high-level plan for solving the 'Lab 15.3 Wordle Solver' lab.

Assistant: To approach the 'Lab 15.3 Wordle Solver' lab, you should consider creating a plan that incorporates both the specifics mentioned in the assignment details and some technical strategies regarding Java development and word verification processes. Here's a structured high-level plan to help you get started on your Wordle Solver program:

1. Initializing the Program

- ****Load Words:**** Begin by reading a list of words from a file - this will work as your initial word bank. You must ensure these words match the Wordle game's constraints, particularly with respect to word length and letter uniqueness based on your selected version (basic or gold). Handle exceptions that might occur while reading the file.

- **Environment Setup:** Initialize necessary data structures such as ArrayLists to manage your list of possible words. Set up any basic structures for the game interface.

2. Game Logic Implementation

- **Word Guessing:** Implement functionality for the user to make guesses. This will involve reading user input from the console.
- **Validation:** Ensure that guesses are valid both in length and in content (i.e., restricting input to alphabetic characters and correct length). If a guess is invalid, prompt the user again until a valid guess is made.

3. Feedback Processing

- **Feedback Interpretation:** After each guess, your program should accept feedback usually in a form indicating right or wrong placements ('Green' for correct and 'Yellow' for present but misplaced).
- **Update Possible Words:** Adjust the list of possible words based on feedback. This is crucial and will involve:
 - Removing words that contain letters marked as 'not present'.
 - Ensuring words have letters in positions flagged as 'correct'
 - Reflecting on feedback of 'present but misplaced' letters to refine possible choices.

4. Reducing Word List

- Continue refining the list of possible words by repeating the feedback loop after each guess. Use a process of elimination based on the feedback provided for each guess to zero in on the correct word.

5. Game End Check

- **Termination Conditions:** Decide termination logic such as the game ends when only one word remains or after a certain number of guesses.
- **Output Final Word:** When the list is narrowed down to one, or another end condition is met, print the final guessed word or indicate failure to deduce the word in the given attempts.

6. Additional Features (For expanded functionality)

- **Repeated Play:** Allow users to play multiple rounds without restarting the program.
- **Difficulty Levels:** Implement different difficulty settings which might include a greater repertoire of words or altered feedback mechanisms.

Keep in mind that each step involves conditional checks and potentially interacting with different components of your program architecture (like UI and word processing logic). As a stretch goal, consider implementing patterns for better code organization such as MVC (Model-View-Controller) if you find your codebase getting complex. This could help manage different aspects of the game distinctly (like handling user input separately from game logic). Remember, the key to a successful implementation is iteratively testing and refining each part of your program.

Evaluation:

The high-level plan provides an accurate overview of the steps needed for the Wordle

Solver. The plan also does provide a bit on what logic would need to be implemented at each step. The plan is helpful since the plan provides only the necessary logic for each step and not any code or pseudocode, allowing the student to figure out how to implement it themselves. This response got a 5 in both accuracy and helpfulness.

5. Discussion:

The results showed that LLM chatbots are mostly accurate for high level plans and pseudocode, but are more error-prone when it comes to debugging. The sample solutions helped the LLM create plans and pseudocode, however did not improve the LLM's ability to read new code and analyze it effectively. The plans and pseudocode often mirrored the sample solution, indicating that the accuracy seen from the results is due to the removal of the need to solve the problem itself and instead translate an existing solution. LLMs struggle with logic the most and providing solutions to it seems to decrease the amount of hallucinations it has due to logic. However, the sample solutions often didn't have an explanation so the chatbot tended to struggle with interpreting when differences in code lead to a meaningful difference in the program.

6. Conclusion:

This study explored the use of LLMs within computer science course education, providing the model with sample solutions to see if sample solutions can assist in the accuracy of the model. The results showed that its high-level plans and pseudocode were correct most of the time, however, the chatbot struggled to analyze new code and compare it with the sample solution correctly. It may be possible to add an explanation to the code so that the model can understand what errors would occur if a certain part is missing or altered for debugging purposes.

7. References:

- [1] Bonner, E., Lege, R., & Frazier, E. (2023). LARGE LANGUAGE MODEL-BASED ARTIFICIAL INTELLIGENCE IN THE LANGUAGE CLASSROOM: PRACTICAL IDEAS FOR TEACHING. *Teaching English with Technology*, 2023(1). <https://doi.org/10.56297/bkam1691/wieo1749>
- [2] Gan, W., Qi, Z., Wu, J., & Lin, C.-W. (2023). *Large Language Models in Education: Vision and Opportunities*. <https://arxiv.org/pdf/2311.13160>
- [3] Hellas, A., Leinonen, J., Sarsa, S., Koutcheme, C., Kujanpää, L., & Sorva, J. (2023). Exploring the Responses of Large Language Models to Beginner Programmers' Help Requests. *ArXiv*. <https://doi.org/10.1145/3568813.3600139>
- [4] Kumar, H., Musabirov, I., Reza, M., Wang, X., Williams, J., Kuzminykh, A., & Liut, M. (2024, January 23). *Impact of Guidance and Interaction Strategies for LLM Use on Learner Performance and Perception*. Arxiv.org. <https://arxiv.org/html/2310.13712v2>
- [5] Zamfirescu-Pereira, J., Qi, L., Hartmann, B., Denero, J., & Norouzi, N. (n.d.). *Conversational Programming with LLM-Powered Interactive Support in an Introductory Computer Science Course*. Retrieved February 26, 2024, from https://gaied.org/neurips2023/files/32/32_paper.pdf