Generating Diverse Code Explanations using the GPT-3 Large Language Model

Conference Paper · August 2022

DOI: 10.1145/3501709.3544280

Stephen Macneil
Temple University

Daniel Mogil
Temple University

Seth Bernstein
Temple University

Erin Ross
Temple University

Generating Diverse Code Explanations using the GPT-3 Large Language Model

Stephen MacNeil
stephen.macneil@temple.edu
Temple University
Philadelphia, PA, USA

Andrew Tran
andrew.tran10@temple.edu
Temple University
Philadelphia, PA, USA

Dan Mogil
daniel.mogil@temple.edu
Temple University
Philadelphia, PA, USA

Seth Bernstein
seth.bernstein@temple.edu
Temple University
Philadelphia, PA, USA

Erin Ross
erinross@temple.edu
Temple University
Philadelphia, PA, USA

Ziheng Huang
z8huang@ucsd.edu
University of California—San Diego
La Jolla, CA, USA

KEYWORDS

large language models, natural language processing, code explanations, computer science education

1 ABSTRACT

Good explanations are essential to efficiently learning introductory programming concepts [10]. To provide high-quality explanations at scale, numerous systems automate the process by tracing the execution of code [8, 12], defining terms [9], giving hints [16], and providing error-specific feedback [10, 16]. However, these approaches often require manual effort to configure and only explain a single aspect of a given code segment. Large language models (LLMs) are also changing how students interact with code [7]. For example, Github's Copilot can generate code for programmers [4], leading researchers to raise concerns about cheating [7]. Instead, our work focuses on LLMs' potential to support learning by explaining numerous aspects of a given code snippet. This poster features a systematic analysis of the diverse natural language explanations that GPT-3 can generate automatically for a given code snippet. We present a subset of three use cases from our evolving design space of *AI Explanations of Code.*

2 USE CASES

To understand the types of explanations GPT-3 [2] can generate, we issued over 700 prompts across numerous code snippets. An example prompt and resulting explanation is shown in Figure 1. We discovered eight explanation types and Figure 2 includes three explanation types to illustrate the explanatory power of GPT-3. The additional types include: 1) tracing the execution of code, 2) fixing bugs and explaining how they were fixed, 3) generating analogies to real world settings, 4) listing relevant programming concepts, and 5) predicting the console output.

ICER 2022, August 7–11, 2022, Lugano and Virtual Event, Switzerland

ACM ISBN 978-1-4503-9195-5/22/08.
https://doi.org/10.1145/3501709.3544280

2.1 Analyzing and explaining time complexity

Instructors rate time complexity as the most difficult programming topic [17]. However, understanding time complexity is important [6, 13] because it facilitates decision-making so students choose an appropriate algorithm for a given problem. This use case shows GPT-3 can identify and explain time complexity.

2.2 Identifying common mistakes made by beginner programmers

Commonality exists in how students solve programming problems [15] and the mistakes they make [1, 11]. Pedagogical techniques, such as the 'muddiest point' highlight these common and most confusing concepts [3, 14]. GPT-3 can automatically create a checklist of common mistakes students might make regarding a given code snippet.

2.3 Summarizing code at multiple levels of abstraction

Before understanding how a code snippet executes, it is often useful to understand the purpose of the code [5]. The summary generated by GPT-3 and shown in Figure 2 defines the goal, traces the execution, and highlights relevant CS concepts such as arrays.

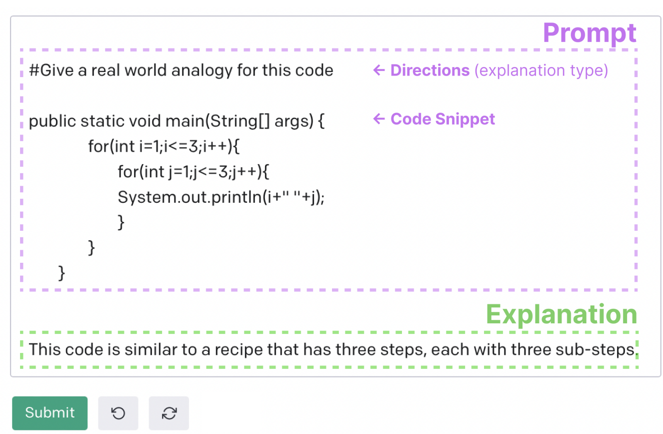

Figure 1: A prompt and explanation based on analogy.

Code Snippet

Given Java Code:

```
int algorithm (int arr[], int l, int r, int x)
 {
   if (r >= l) {

     int mid = l + (r - l) / 2;

     if (arr[mid] == x)
       return mid;

    if (arr[mid] > x)
      return algorithm(arr, l, mid - 1, x);

    return algorithm(arr, mid + 1, r, x);
  }
  return -1;
}
```

1. Time Complexity

What is the time complexity of this code and why?

The time complexity of this code is O(log n). This is because the code is using a Binary Search algorithm, which halves the search space each time it is called.

2. Identifying Common Mistakes

List the common mistakes a beginner programmer might make while writing this code. How can a beginner programmer avoid each of these mistakes?

A beginner programmer might make the following mistakes while writing this code:

1. Not understanding how binary search works - Binary search is a search algorithm that searches for an element in a sorted array by repeatedly dividing the array in half and comparing the element to the middle element of the array. If the element is not found, the search continues in the lower or upper half of the array, depending on whether the element is less than or greater than the middle element, respectively.

2. Not correctly implementing the binary search algorithm - In order to correctly implement binary search, the programmer must ensure that the array is sorted before beginning the search. Furthermore, the programmer must correctly handle the case when the element is not found in the array.

3. Not using the proper data structures - Binary search can only be performed on an array. If the programmer is using a linked list, for example, binary search cannot be used.

4. Not using the proper variable types - Binary search requires that the array be of type int. If the programmer is using a different data type, binary search will not work.

5. Not understanding the time complexity of binary search - Binary search has a time complexity of O(log n), which means that it is a very efficient algorithm. If the programmer is not aware of this, they may try to improve the efficiency of the algorithm by making unnecessary changes.

3. Summarizing Code

Summarize and explain the goal of this code.

The goal of this code is to find the index of a given element in an array using a binary search algorithm. The code takes in an array, the left and right indices of the array, and the element to be found as parameters. If the element is found, the code returns the index of the element. If the element is not found, the code returns -1.

Figure 2: Three example explanations automatically generated by GPT-3 for an 'anonymized' Binary Search code snippet.

3 DISCUSSION

Our three use cases demonstrate the potential for GPT-3 to explain code for intro CS students. Our poster presentation will feature all eight explanation types as a design space of explanations to convey the diversity of explanations that can be generated by LLMs. We will highlight best practices for generating effective explanations and pitfalls that lead to less effective explanations. We are evaluating the usefulness of these explanations in a series of summer classes.

REFERENCES

[1] Amjad Altadmri and Neil CC Brown. 37 million compilations: Investigating novice programming mistakes in large-scale student data. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, pages 522–527, 2015.
[2] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in Neural Information Processing Systems*, 33:1877–1901, 2020.
[3] Adam Carberry, Stephen Krause, Casey Ankeny, and Cynthia Waters. "unmuddying" course content using muddiest point reflections. In *2013 IEEE Frontiers in Education Conference (FIE)*, pages 937–942. IEEE, 2013.
[4] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.
[5] Kathryn Cunningham, Yike Qiao, Alex Feng, and Eleanor O'Rourke. Bringing "high-level" down to earth: Gaining clarity in conversational programmer learning goals. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education V. 1*, SIGCSE 2022, page 551–557, New York, NY, USA, 2022. Association for Computing Machinery.
[6] Elvina Elvina and Oscar Karnalim. Complexitor: An educational tool for learning algorithm time complexity in practical manner. *ComTech: Computer, Mathematics and Engineering Applications*, 8(1):21–27, 2017.
[7] James Finnie-Ansley, Paul Denny, Brett A. Becker, Andrew Luxton-Reilly, and James Prather. The robots are coming: Exploring the implications of openai codex on introductory programming. In *Australasian Computing Education Conference*, ACE '22, page 10–19, New York, NY, USA, 2022. ACM.
[8] Philip J Guo. Online python tutor: embeddable web-based program visualization for cs education. In *Proceeding of the 44th ACM technical symposium on Computer science education*, pages 579–584, 2013.
[9] Andrew Head, Codanda Appachu, Marti A Hearst, and Björn Hartmann. Tutorons: Generating context-relevant, on-demand explanations and demonstrations of online code. In *2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 3–12. IEEE, 2015.
[10] Samiha Marwan, Ge Gao, Susan Fisk, Thomas W. Price, and Tiffany Barnes. Adaptive immediate feedback can improve novice programming engagement and intention to persist in computer science. In *Proceedings of the 2020 ACM Conference on International Computing Education Research*, ICER '20, page 194–203, New York, NY, USA, 2020. Association for Computing Machinery.
[11] Davin McCall and Michael Kölling. Meaningful categorisation of novice programmer errors. In *2014 IEEE Frontiers in Education Conference (FIE) Proceedings*, pages 1–8. IEEE, 2014.
[12] Greg L Nelson, Benjamin Xie, and Amy J Ko. Comprehension first: evaluating a novel pedagogy and tutoring system for program tracing in cs1. In *Proceedings of the 2017 ACM conference on international computing education research*, pages 2–11, 2017.
[13] Miranda Parker and Colleen Lewis. What makes big-o analysis difficult: understanding how students understand runtime analysis. *Journal of Computing Sciences in Colleges*, 29(4):164–174, 2014.
[14] Daniel Perez, Leila Zahedi, Monique Ross, Jia Zhu, Tiffany Vinci-Cannava, Laird Kramer, and Maria Charters. Wip: An exploration into the muddiest points and self-efficacy of students in introductory computer science courses. In *2020 IEEE Frontiers in Education Conference (FIE)*, pages 1–5. IEEE, 2020.
[15] Chris Piech, Mehran Sahami, Jonathan Huang, and Leonidas Guibas. Autonomously generating hints by inferring problem solving policies. In *Proceedings of the second (2015) acm conference on learning@ scale*, pages 195–204, 2015.
[16] Thomas W Price, Yihuan Dong, and Dragan Lipovac. isnap: towards intelligent tutoring in novice programming environments. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on computer science education*, pages 483–488, 2017.
[17] Carsten Schulte and Jens Bennedsen. What do teachers teach in introductory programming? In *Proceedings of the second international workshop on Computing education research*, pages 17–28, 2006.