5-2023

Navigating complexity in software engineering: a prototype for comparing GPT-n solutions

Christoph TREUDE
Singapore Management University, ctreude@smu.edu.sg

Navigating Complexity in Software Engineering: A Prototype for Comparing GPT-n Solutions

Christoph Treude
School of Computing and Information Systems
The University of Melbourne
Melbourne, Australia
christoph.treude@unimelb.edu.au

***Abstract*—Navigating the diverse solution spaces of non-trivial software engineering tasks requires a combination of technical knowledge, problem-solving skills, and creativity. With multiple possible solutions available, each with its own set of trade-offs, it is essential for programmers to evaluate the various options and select the one that best suits the specific requirements and constraints of a project. Whether it is choosing from a range of libraries, weighing the pros and cons of different architecture and design solutions, or finding unique ways to fulfill user requirements, the ability to think creatively is crucial for making informed decisions that will result in efficient and effective software. However, the interfaces of current chatbot tools for programmers, such as OpenAI's ChatGPT or GitHub Copilot, are optimized for presenting a single solution, even for complex queries. While other solutions can be requested, they are not displayed by default and are not intuitive to access. In this paper, we present our work-in-progress prototype "GPTCompare", which allows programmers to visually compare multiple source code solutions generated by GPT-n models for the same programming-related query by highlighting their similarities and differences.**

***Index Terms*—Chatbots, diversity, complexity, solution spaces**

I. Introduction

Choosing the right solution to a programming problem is a complex task that depends on a variety of factors, such as project requirements, development team skillset, project deployment environment, and desired level of security, performance, and maintainability. For example, an important consideration is whether to rely on external libraries or APIs [1]. Although using these can save time and effort in the short term, they can also introduce security vulnerabilities and increase the risk of dependency issues [2]. Another example is the trade-off between readability and maintainability versus performance [3]. Code that is easy to read and maintain is generally preferred, but in real-time systems, for example, high performance is essential and it may be worth sacrificing some readability and maintainability.

When programmers write source code by hand, they are aware of trade-offs and can consider them before writing a new piece of code. However, when artificial intelligence is asked to generate code, the complexity of the trade-offs may be hidden from the programmer, making it difficult to fully understand and evaluate the code that is generated, often with negative consequences. For example, a recent study found that programmers write more insecure code with artificial intelligence assistants, while they are more likely to believe that they wrote secure code [4].

Large language models that power many chatbots for programmers can generate a very large number of solutions for a given task, but current user interfaces may not always provide enough information about these solutions or allow for effective exploration of solution spaces [5]. Instead of informing programmers proactively about the fact that multiple solutions exist and what their pros and cons are, many chatbots simply offer the single solution with the highest probability given the user's input. This can lead programmers to accept the solution presented to them without fully understanding the implications of the code and the differences that other solutions might offer, a phenomenon known as the order effect in recommender systems [6].

In this paper, we argue that it is possible to modify the user interface of a chatbot for programmers in such a way that it shows multiple solutions to a given task and highlights their commonalities and differences. To demonstrate the feasibility of this approach, we introduce an initial prototype of a chatbot for programmers that shows multiple solutions and visually highlights their similarities and differences. This prototype serves as a proof-of-concept and demonstrates that it is possible to create a user interface that provides more information about the different solutions that are generated by artificial intelligence.

II. Motivating Example

Figure 1 shows an example of the code generated by GPT-3[1] in response to the prompt "Write a Python function that returns the largest element in a list."[2] The code snippet defines a function called `getMax` that takes a single argument, a list. The function starts by initializing a variable called `maxValue` to the value of the first element in the list. It then iterates through the remaining elements in the list, comparing each element with the current `maxValue`. If the current element is larger than `maxValue`, the `maxValue` is updated to the value of the current element.

The code snippet also includes an example of how to use the `getMax` function. It calls the function with an example list

[1]https://beta.openai.com/playground

[2]All example tasks used in this paper are taken from https://adriann.github.io/programming_problems.html.

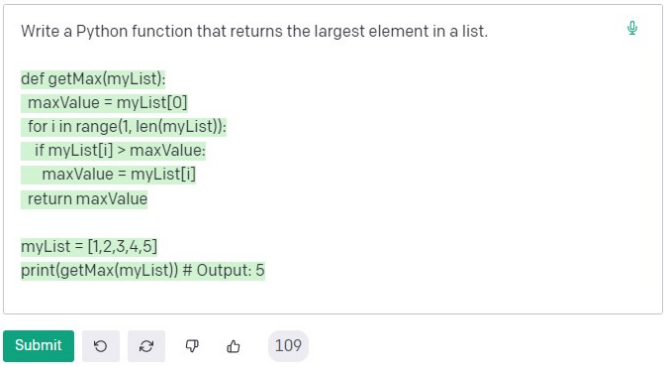

Fig. 1. First code snippet generated by GPT-3 for the prompt "Write a Python function that returns the largest element in a list."

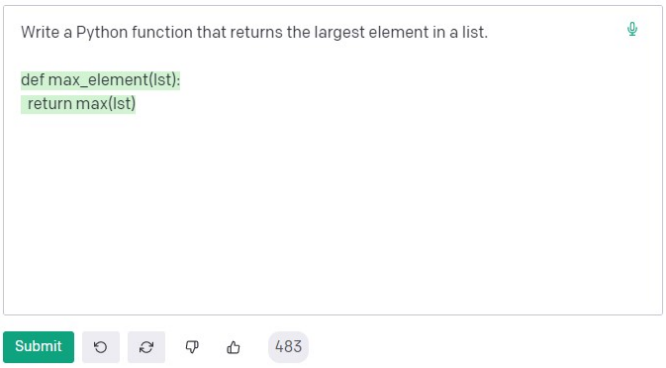

Fig. 2. Second code snippet generated by GPT-3 for the prompt "Write a Python function that returns the largest element in a list."

containing five elements and provides a source code comment that specifies the expected output of the function.

It is notable that the code does not make use of any external APIs or libraries, and instead re-implements the functionality for determining the maximum value in a list from scratch. This approach has the advantage of being simple to understand, but may not be as efficient as using a built-in function or library, depending on the size of the input list. Additionally, this approach does not take advantage of code reuse, which could make it less maintainable in the long run.

Upon closer inspection of the GPT-3 interface, it becomes apparent that users can regenerate the response if they are not satisfied with the first response by pressing the button with the two arrows to the right of the `Submit` button. Pressing this button generates a new code snippet, as shown in Figure 2.

The code snippet provided in Figure 2 is much shorter than the one provided in Figure 1. The method name has changed from `getMax` to `max_element`, the name of the argument has changed from `myList` to `lst`, but the more significant change is that this code snippet simply uses Python's built-in `max` function to achieve its task. This approach is more efficient and takes advantage of code reuse, unlike the first snippet. However, unlike the first snippet, it does not provide example input and output.

This example also highlights the limitations of the current GPT-3 interface in terms of navigating multiple solutions. Even with this small programming task, the interface does not provide an easy way to compare different solutions side-by-side, as the old snippet disappears when a new one is generated. The example also demonstrates that even with trivial programming tasks, different ways of solving them exist, in this case with or without reliance on built-in Python functions.

Figure 3 illustrates the output of our prototype GPTCompare for the same prompt. Our prototype, by default, displays the first five solutions generated by GPT-3 for a given prompt, but it also adds additional highlighting to indicate the commonalities and differences between the solutions. This allows programmers to easily identify which elements are shared across all solutions and which elements are unique to specific solutions.

In the provided example, all solutions start with the Python keyword `def`, indicating that they all define a function. Furthermore, all solutions contain an argument whose name starts with `list`. All solutions contain a `return` statement, indicating that they all produce a value. These common elements are shown in black.

On the other hand, elements that are not common to all solutions are highlighted with different shades of blue (by default), indicating their uniqueness. For example, the source code comment in the last line (`Output: 98`) is unique and does not appear in any of the other solutions. It is therefore shown in a bright shade of blue, making it stand out. On the other hand, the call to the built-in `max` function is less unique in this set of solutions, as both the third and fourth solutions rely on this function to achieve their task. It is shown in a darker shade of blue, indicating that it is not as unique as the first example.

III. Implementation

Figure 4 illustrates the output of GPTCompare in a simple scenario. The prompt given to GPT-3 was "Write a Python program that prints 'Hello World' to the screen." GPT-3 generated almost identical code snippets for this prompt, with the exception of one snippet that was preceded by a short source code comment and an exclamation mark added to the end of the string passed to the `print` function. Upon closer inspection, it is apparent that GPT-3's generated code snippets were not consistent in their use of single quotes or double quotes for strings. Two of the five solutions use double quotes, while the remaining three use single quotes. GPTCompare utilizes color highlighting to distinguish these differences among solutions. The source code comment for the third solution is highlighted in a bright blue color, as is the exclamation mark in the same solution. The different types of quotation marks are highlighted in a darker shade of blue, indicating that they are less unique as they appear in two or three of the solutions, respectively.

Algorithm 1 describes the process used by GPTCompare to highlight the differences and similarities between multiple

```
def largest_element(list): large = list[0] for i in list: if i > large: large = i return large lst = [1, 3, 5, 4, 2] print(largest_element(lst))

def largest_element_in_list(list_of_elements): current_largest = list_of_elements[0] for element in list_of_elements: if element > current_largest:
current_largest = element return current_largest

def get_max(list): return max(list)

def largest_element(list): return max(list)

def max_element(list): max = 0 for x in list: if x > max: max = x return max list = [1, 4, 7, 12, 98, 34] print(max_element(list)) #Output: 98
```

Fig. 3. Output generated by GPTCOMPARE for the prompt "Write a Python function that returns the largest element in a list."

```
print("Hello World")

print('Hello World')

# Python Program print("Hello World!")

print('Hello World')

print('Hello World')
```

Fig. 4. Output generated by GPTCOMPARE for the prompt "Write a Python program that prints 'Hello World' to the screen."

```
inputs ← GPT-n.responses;
n ← inputs.length;
for i ← 1, n do
    diffs ← [];
    for j ← 1, n do
        diff ← compare(inputs[i], inputs[j]);
        diffs.append(diff);
    end
    m ← diffs[0].length;
    for k ← 1, m do
        uniqueness ← 0;
        foreach diff ∈ diffs do
            if diff[k].startswith(+) then
                uniqueness += 1;
            end
        end
        if uniqueness > 0 then
            output += diff[k]
            (color: 127 + uniqueness * 32);
        else
            output += diff[k];
        end
    end
end
```

Algorithm 1: GPTCOMPARE's algorithm

code snippets. It begins by using a diff algorithm such as `difflib`[3] to calculate the differences between each pair of input code snippets. In the next phase, the algorithm goes through the output character-by-character and calculates the extent to which each character is unique based on the output of the diff algorithm. Using five code snippets as a starting point, the uniqueness of each character is a number between 0 and 4: If the character also exists in all other solutions, its uniqueness is 0. If it is unique compared to all other solutions (i.e., if it was recognized as an added character in all relevant diffs), its uniqueness is 4. If it was recognized as an added character in only some of the relevant diffs, its uniqueness would be 1, 2, or 3. The color used to display each character is then calculated based on the 256 levels of an RGB color model: if uniqueness is 0, the color will be 0, if uniqueness is larger than 0, the color will be at least 127 (half of the RGB scale) plus a factor multiplied by the uniqueness, for a maximum level equal to the maximum of the RGB scale.

[3]https://docs.python.org/3/library/difflib.html

IV. RELATED WORK

In this section, we review related work on user interfaces for software bots and on using artificial intelligence to generate source code. Our work lies at the intersection of these research areas.

A. User interfaces for software bots

A software bot is "a conduit or an interface between users and services, typically through a conversational user interface" [7]. Historically, research on bots for programmers has focused mainly on bots that proactively interact with programmers, such as code review bots integrated into frameworks such as GitHub Actions [8]. Research has suggested that these bots should aim to provide clear, concise, and well-organized information and focus on appropriate ways of presenting information [9]. Other efforts have focused on the use of bots to help programmers write queries and questions [10], [11] and the reduction of the number of steps required to integrate code snippets from online sources into a programmer's code base [12], [13].

However, less attention has been paid to the user interfaces of such bots for programmers. In our previous work, we have suggested that tools such as GitHub Copilot should prioritize the provision of diverse suggestions rather than redundant ones and explore different methods to highlight commonalities and differences between recommendations [5]. GPTCOMPARE provides a preliminary investigation of these ideas, which are supported by related work from other domains. For example, diversifying reply suggestions for instant messaging systems [14] and emails [15] has been shown to improve creativity [16].

B. Using artificial intelligence to generate code

The application of modern natural language processing techniques to programming languages can be traced back to the idea that software is a form of human communication [17], known as the naturalness hypothesis [18]. This hypothesis has led to the successful application of natural language processing techniques to source code, such as the adaptation of machine translation techniques to translate source code across different programming languages [19], the automated generation of documentation for source code [20], or the auto-completion of code [21]. These advances are enabled by models such as PLBART [22], CodeBERT [20], and Codex [21], the latter of which forms the basis of GitHub's Copilot [23].

Although these natural language processing techniques have the potential to greatly improve the efficiency and effectiveness of software development, there are concerns regarding their use. One problem with large language models is that they can sometimes "hallucinate" [24]. A recent study found that a state-of-the-art model was more likely to generate code containing a vulnerability if the query asked for code without that vulnerability [25]. Another study found that programmers with artificial intelligence assistants were more likely to believe that they wrote secure code, despite having more insecure code [4]. These findings highlight the need for further research on the interface between programmers and the capabilities of large language models, such as GPT-3.

V. Research Agenda

User interface improvements: The initial prototype of GPTCompare has demonstrated the potential to modify the user interface of a chatbot to make it easier for programmers to compare different solutions generated by artificial intelligence. However, further improvements are needed to make it more user-friendly and effective for programmers to use in their daily work. One of the main challenges is to improve the layout of the solutions, as the current prototype does not take into account line breaks. Color settings need to be optimized to help programmers distinguish between unique, common, and essential aspects of the solutions. Another important aspect to consider is the integration of GPTCompare into a code editor, since the current version is a stand-alone prototype.

Unit of comparison: An important avenue for future research in tools like GPTCompare is to explore the most effective unit of comparison. In the current prototype, we chose to use character-level comparison, but there are other options that could bring additional benefits. For example, a token-level comparison could ignore simple variable renaming and make the comparison more meaningful. Additionally, depending on the size and complexity of the solutions being compared, line-level or function-level comparisons might be appropriate. Related research on source code representation [26] has also considered methods such as control flow graphs and program dependence graphs. These approaches could potentially bring even more benefits to the comparison process, but they also present challenges in terms of visual representation and usability for programmers. The most suitable unit of comparison is likely going to depend on programming language and task, as well as user preferences.

Interactivity: Future research in code generation should focus not only on creating individual solutions but also on developing interactive tools that allow programmers to seamlessly integrate different aspects of generated code snippets into new, customized solutions. This would eliminate the need for tedious and error-prone manual copy-and-paste, as well as the potential for bugs that can arise from manual integration. The small example in Figures 1 and 2 highlights the potential benefits of such interactivity. Currently, there is no tool support to combine the implementation of the function in Figure 2 with the input and output example in Figure 1. As the complexity of the code increases, the potential for errors also increases.

Other software engineering tasks: Large language models have the potential to not only generate code, but also to assist in debugging by providing explanations for error messages and suggesting fixes. A recent study [27] found that while Codex-produced explanations were considered "quite comprehensible", they were only correct in about half of all cases. Debugging, like code generation, can be a complex task that requires exploring solution spaces. Displaying a single solution or explanation may not be sufficient in many complex scenarios. Therefore, future research should focus on developing tools that assist programmers in effectively navigating the solution space and identifying the most appropriate solutions.

Explicit trade-offs: Providing programmers with the ability to navigate solution spaces is a necessary but not sufficient step towards ensuring that they are able to make informed decisions about the trade-offs associated with different solutions. Future research should also focus on developing tools that provide programmers with additional information that can help them understand and navigate these trade-offs. One approach could be to augment generated solutions with relevant information such as security and performance metrics, possibly in combination with visualizations.

Reinforcement learning: It is unlikely that all programmers would have the same preferences for code snippets, even in similar contexts. To address this, chatbots aimed at assisting programmers in exploring solution spaces could use reinforcement learning to learn from the preferences and behavior of their users. This approach would allow the chatbot to adapt to both the overall preferences of the general user population and the individual preferences of specific users.

Evaluation: Evaluating the effectiveness of tools such as GPTCompare requires conducting user studies to understand how programmers interact with existing tools, as well as the approaches proposed in this paper. Different types of users, such as novice programmers and experienced programmers, will likely have different needs and use technology differently depending on their background and level of expertise. User studies can provide valuable insights into the usability and utility of the tools, identify areas for improvement, and tailor the tools to meet the needs of different groups, ultimately helping programmers navigate the complexity of software engineering tasks.

References

[1] E. Larios Vargas, M. Aniche, C. Treude, M. Bruntink, and G. Gousios, "Selecting third-party libraries: The practitioners' perspective," in *Proceedings of the Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 245–256.

[2] N. Zahan, T. Zimmermann, P. Godefroid, B. Murphy, C. Maddila, and L. Williams, "What are weak links in the npm supply chain?" in *Proceedings of the International Conference on Software Engineering: Software Engineering in Practice*, 2022, pp. 331–340.

[3] L. Traini, D. Di Pompeo, M. Tucci, B. Lin, S. Scalabrino, G. Bavota, M. Lanza, R. Oliveto, and V. Cortellessa, "How software refactoring impacts execution time," *ACM Transactions on Software Engineering and Methodology*, vol. 31, no. 2, pp. 1–23, 2021.

[4] N. Perry, M. Srivastava, D. Kumar, and D. Boneh, "Do users write more insecure code with AI assistants?" *arXiv preprint arXiv:2211.03622*, 2022.

[5] C. Treude, "Taming multi-output recommenders for software engineering," in *Proceedings of the International Conference on Automated Software Engineering*, 2022, pp. 1–5.

[6] X. Guo, L. Wang, M. Zhang, and G. Chen, "First things first? Order effects in online product recommender systems," *ACM Transactions on Computer-Human Interaction*, 2022.

[7] M.-A. Storey and A. Zagalsky, "Disrupting developer productivity one bot at a time," in *Proceedings of the International Symposium on Foundations of Software Engineering*, 2016, pp. 928–931.

[8] T. Kinsman, M. Wessel, M. A. Gerosa, and C. Treude, "How do software developers use GitHub Actions to automate their workflows?" in *Proceedings of the International Conference on Mining Software Repositories*, 2021, pp. 420–431.

[9] M. Wessel, A. Zaidman, M. A. Gerosa, and I. Steinmacher, "Guidelines for developing bots for GitHub," *IEEE Software*, 2022.

[10] M. Liu, X. Peng, A. Marcus, C. Treude, J. Xie, H. Xu, and Y. Yang, "How to formulate specific how-to questions in software development?" in *Proceedings of the Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 306–318.

[11] K. Cao, C. Chen, S. Baltes, C. Treude, and X. Chen, "Automated query reformulation for efficient search based on query logs from Stack Overflow," in *Proceedings of the International Conference on Software Engineering*, 2021, pp. 1273–1285.

[12] B. A. Campbell and C. Treude, "NLP2Code: Code snippet content assist via natural language tasks," in *Proceedings of the International Conference on Software Maintenance and Evolution*, 2017, pp. 628–632.

[13] B. Reid, M. d'Amorim, M. Wagner, and C. Treude, "NCQ: Code reuse support for Node.js developers," *arXiv preprint arXiv:2101.00756*, 2021.

[14] B. Deb, P. Bailey, and M. Shokouhi, "Diversifying reply suggestions using a matching-conditional variational autoencoder," in *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2*, 2019, pp. 40–47.

[15] D. Buschek, M. Zürn, and M. Eiband, "The impact of multiple parallel phrase suggestions on email input and composition behaviour of native and non-native English writers," in *Proceedings of the Conference on Human Factors in Computing Systems*, 2021, pp. 1–13.

[16] N. Singh, G. Bernal, D. Savchenko, and E. L. Glassman, "Where to hide a stolen elephant: Leaps in creative writing with multimodal machine intelligence," *ACM Transactions on Computer-Human Interaction*, 2022.

[17] J. Sun, Q. V. Liao, M. Muller, M. Agarwal, S. Houde, K. Talamadupula, and J. D. Weisz, "Investigating explainability of generative AI for code through scenario-based design," in *Proceedings of the International Conference on Intelligent User Interfaces*, 2022, pp. 212–228.

[18] A. Hindle, E. T. Barr, M. Gabel, Z. Su, and P. Devanbu, "On the naturalness of software," *Communications of the ACM*, vol. 59, no. 5, pp. 122–131, 2016.

[19] B. Roziere, M.-A. Lachaux, L. Chanussot, and G. Lample, "Unsupervised translation of programming languages," *Advances in Neural Information Processing Systems*, vol. 33, pp. 20 601–20 611, 2020.

[20] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "Codebert: A pre-trained model for programming and natural languages," *arXiv preprint arXiv:2002.08155*, 2020.

[21] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[22] W. U. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "Unified pre-training for program understanding and generation," *arXiv preprint arXiv:2103.06333*, 2021.

[23] N. Nguyen and S. Nadi, "An empirical evaluation of GitHub Copilot's code suggestions," in *Proceedings of the International Conference on Mining Software Repositories*, 2022, pp. 1–5.

[24] Z. Ji, N. Lee, R. Frieske, T. Yu, D. Su, Y. Xu, E. Ishii, Y. Bang, A. Madotto, and P. Fung, "Survey of hallucination in natural language generation," *ACM Computing Surveys*, 2022.

[25] H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri, "Asleep at the keyboard? Assessing the security of GitHub Copilot's code contributions," in *Proceedings of the Symposium on Security and Privacy*, 2022, pp. 754–768.

[26] Y. Jiang, X. Su, C. Treude, and T. Wang, "Hierarchical semantic-aware neural code representation," *Journal of Systems and Software*, vol. 191, p. 111355, 2022.

[27] J. Leinonen, A. Hellas, S. Sarsa, B. Reeves, P. Denny, J. Prather, and B. A. Becker, "Using large language models to enhance programming error messages," *arXiv preprint arXiv:2210.11630*, 2022.