# The Behavior of Large Language Models When Prompted to Generate Code Explanations

**Priti Oli** *
University of Memphis
Memphis TN 38152,USA
poli@memphis.edu

**Rabin Banjade** *
University of Memphis
Memphis TN 38152,USA
rbnjade1@memphis.edu

**Jeevan Chapagain**
University of Memphis
Memphis TN 38152,USA
jchpgain@memphis.edu

**Vasile Rus**
University of Memphis
Memphis TN 38152,USA
vrus@memphis.edu

## Abstract

This paper systematically investigates the generation of code explanations by Large Language Models (LLMs) for code examples commonly encountered in introductory programming courses. Our findings reveal significant variations in the nature of code explanations produced by LLMs, influenced by factors such as the wording of the prompt, the specific code examples under consideration, the programming language involved, the temperature parameter, and the version of the LLM. However, a consistent pattern emerges for Java and Python, where explanations exhibit a Flesch-Kincaid readability level of approximately 7-8 grade and a consistent lexical density, indicating the proportion of meaningful words relative to the total explanation size. Additionally, the generated explanations consistently achieve high scores for correctness, but lower scores on three other metrics: completeness, conciseness, and specificity.

## 1 Introduction

Explanation of code examples is an effective instructional tool to help students in intro-to-programming courses master programming concepts and develop other skills such as code comprehension skills [2, 12]. Asking students to self-explain (a more active learning strategy) or simply reading worked-out examples accompanied by explanations provided by experts (a more passive learning strategy) was shown to have a positive impact on learning gains and code comprehension skills [12, 19]. Furthermore, interactive learning strategies such as scaffolded self-explanations while interacting with a human or computer tutor have also been explored with very positive results [19, 12]. Many types of explanations have been explored, such as 'Explain-in-plain-English' [10, 11, 21] or 'stepwise explanation' [15] or code comprehension-theory driven explanations [19]. In all those instances, experts are tasked to create corresponding explanations of the target code examples, which is tedious and expensive. Large Language Models (LLMs; [4]) have been recently explored as a potential solution to alleviate those authoring costs associated with expert-generated explanations. Automating the task of code explanation generation could lead to tremendous advantages in terms of scaling up the use of explanations across topics and domains.

LLMs are impressive technologies relative to what has been done before in terms of content generation; however, there are several serious challenges with LLMs, such as non-deterministic behavior,

*Both authors contributed equally.

NeurIPS'23 Workshop on Generative AI for Education (GAIED).

generating incorrect output [1] or so-called hallucinogenic behavior (generating untrue facts, e.g., person X won award Y in year Z when in fact person W won that award), and data contamination (test data being very similar or identical to training data which can be considered some form of memorization; [3]). Those challenges suggest a more cautious approach to recommending or using such tools without proper, solid, systematic studies that can document the strengths and weaknesses of LLMs. Studies published recently with respect to generating code explanations (e.g., code summaries, line-by-line explanations; [15, 7, 9]) do not report important aspects of their use of LLMs, such as the actual prompts used or, if they explored a number of prompts and other parameters' values before finding the best combination of parameters that generated the kind of output wanted. Details of the prompt engineering and parameter space exploration process is not reported comprehensively.

In response to these challenges and the need for a more systematic exploration of LLMs to better understand their behavior, we conduct a systematic exploration of the space of input parameters and an analysis of the behavior of LLMs focusing on the task of generating code explanations for code examples of the type used in intro-to-programming courses. Specifically, we study how five major input parameters alter the output of the LLMs, i.e., how the generated code explanations vary while those input parameters vary. We focused on the following five input parameters: input prompt wording, code example type, temperature, LLM model, and programming language.

## 2 Related Work

Relevant to our work, LLMs have been used and studied extensively so far from two major perspectives: (1) generating code from natural language descriptions (see NL2CODE survey paper for a good overview [14, 5, 22]) and (2) generating explanations of code examples. Due to space reasons, we will focus on the latter.

Sarsa and colleagues [15] used LLMs to generate programming exercises (including sample solutions and test cases) and code explanations and assessed these items qualitatively and quantitatively. For the explanations part, they considered three types of explanations: a high-level description of the code, a problem statement-like description of the code, and a step-by-step explanation of the code. However, their work focused on step-by-step explanations of code. They used this prompt, "Step-by-step explanation of the above program," and noted that this prompt 'tended' to produce line-by-line explanations. The word 'tended' hints at some inconsistent behavior, which our systematic study reported here confirms. They set the temperature to zero as they wanted 'precise explanations instead of creative ones'. Sarsa and colleagues report 67% correctness of the generated explanations and 90% coverage, i.e., 90% of 'all parts of the code' were explained. They used in their study "a small set of exercises that have been featured in computing education research and that are often used in the teaching contexts of the researchers." On the one hand, using such widely used exercises lends more credibility and allows comparison to prior work; on the other hand, using well-known exercises for studying LLMs poses a major issue: data contamination. Due to space reasons, we do not address the issue of data contamination in this paper but plan to explore it and report on it in future work.

McNeil and colleagues [9] generated three types of code explanations using LLMs and integrated them into an interactive e-book used in a web software development course. The three types of explanations generated were a line-by-line explanation, a list of important concepts, and a high-level summary of the code. They report using the default parameter for GPT-3 Davinci model and, importantly, prompts sent to the two LLMs, GPT-3 and Codex, are different (shown in their paper in Figure 1): "Summarize and explain this code snippet" versus "Summarize and explain the goal of the above code." The latter only asks for a summary and explanation of the goal of the code. They claim, 'Based on our preliminary findings, explanations appear helpful for learning'. Also, they report that "at a cursory glance, we did not observe any significant mistakes, and the explanations were, in general, correct (although at times omitting details, as one would expect)." this is partly due to prompt engineering without providing details of the exact prompt engineering process.

MacNeil and colleagues used GPT-3 to generate 'diverse ' explanations [7]. They explored what types of explanations GPT-3 can generate. Furthermore, the type of explanations was primed by the authors by the nature of the prompts they sent to the LLM. Notably, some of those prompts do not use the word 'explanation' (or its morphological variations or closely related phrases such as

'line-by-line explanation'). One example prompt they used is 'Give a real analogy for this code.' Besides the fact that many of their prompts didn't use 'explanation' keywords, the 'diverse' term in the title can also be ambiguous in the sense that the diversity of explanations can be defined in very different ways. For instance, Maharjan and colleagues [8] shows how linguistically diverse student explanations are, varying from just one word to sophisticated paraphrases of expert explanations.

In sum, prior work on using LLMs to generate code explanations is limited in several ways: they have a wide range of views of what a code explanation should be, they do not always report the exact prompts they used, they do not report other important parameters, or do not vary such parameters systematically (e.g., they use the default value of the temperature parameter), nor analyze how such other important factors may alter the nature of the explanations generated.

## 3 Methodology

We detail in this section the methodology we used to systematically explore the space of input parameters for LLMs and the prompt engineering process.

### 3.1 Code Explanation Generation Across various factors

Prompting LLMs is probably the most important factor, as this is the input based on each, the LLMs will produce the output; it is also the trickiest factor as the input prompt can alter the behavior/output of the LLMs substantially, e.g., prompts can be very simple requests such as P1 in Table 1 *Can you explain this code?* to heavily contextualized prompts (see prompts C1 and C2 in the table) where the user is providing a rich paragraph of contextualized information such as *You are a tutor who is supposed to teach students programming. The students are novices ...* (see the table for the full prompt) to iterative prompts to a few shot prompting in which examples of code and explanations are given to the LLMs to self-prompts, i.e., LLM-generated prompts - the user creates a simple prompt asking the LLM to generate a prompt for LLMs which in turn will be used to prompt the LLM.

This paper focuses on simple and contextualized prompts such as those shown in Table 1 and how their wording influences the LLMs' behavior. For the wording, we started with simple wording inspired by previous work (see [9]) and then made small, step-wise changes to the wording while also asking for different types of explanations based on code comprehension theories [16] (e.g., generic explain the code, line-by-line, block-level, summary, contextualized, used-by-others).

In addition to prompt wording, we investigate four factors influencing code explanations: language model, temperature, code example complexity (basic, intermediate, advanced), and programming language (Java, Python, C++). The language models we investigated were OpenAI's ChatGPT-3.5-turbo-0613 and chatGPT-4-0613 [13], and Meta's open-source model LLMa2-chat[2] [20]). The temperature parameter can take values between 0 and 2, with values closer to 0, limiting the diversity of the output by limiting how 'generative/creative' the LLM is.

All our code examples are typical examples used in intro-to-programming courses (CS1 and CS2). We chose several examples from very basic to intermediate to more complex. Specifically, we used the following code examples: *calculate the average of numbers, calculate the area of a circle, search a number using binary search, translate a point, and generate a bingo board*, the Java version of which is shown in Appendix 6.1. The code examples vary based on concepts, difficulty, and structure.

### 3.2 The Data: The Generated Code Explanations

We have generated explanations using 13 input prompts, 3 LLMs, 5 different temperature levels (0, 0.5, 1, 1.5, 2.0), 5 code examples, and 3 programming languages for a total of 3,510 code explanations. Explanations with temperature value 2 were discarded in our analysis as they are non-sensical for all 3 LLMs. We set a maximum token limit of 500 for the generated explanations.

[2]https://huggingface.co/meta-llama/Llama-2-7b-chat-hf

| Symbol | Prompt | Comments |
|---|---|---|
| P1 | Can you explain this code? | Simple prompts and variation |
| P2 | Can you self-explain this code? | |
| P3 | Can you explain this code to a learner? | |
| P4 | Can you explain this code to someone learning to program? | |
| P5 | Can you summarize this code? | Prompt for summary |
| P6 | Can you summarize this code for a learner? | |
| P7 | Can you explain this code at statement level? | line by line explanations |
| P8 | Can you explain this code at block level? | logical/functional level explanations |
| P9 | Can you explain this code without breaking down individual statements? | |
| C1 | **Context:** You are a tutor who is supposed to teach students programming. The students are novices and your task is to give students learning tasks. One of the learning tasks is to read and explain code examples. The explanation should clearly articulate what the goal of the code is, what the major functional blocks are, and implementation details. **Prompt:** Given this Java code, explain the code to your students in order to help them understand what the code does and learn the covered programming concepts. | Contextualized |
| C2 | **Context:** You are supposed to read code examples in order to understand them. You will be given one code example at a time, your task is to read each code example as carefully as you can and then explain your understanding of the code as best as you can. **Prompt:** Given this Java code, read the code carefully and explain what it does to potential students who learn programming. | |
| D1 | Explain the following code line by line as a bulleted list: | Prompts used in prior work [9] |
| D2 | Give a detailed explanation of the purpose of the following code: | |
| D3 | Summarize and explain the goal of the above code | |

Table 1: The input prompts used from simple to contextualized.

## 4 Results

We analyzed the generated explanations using a number of criteria, including readability, lexical diversity, and correctness, as well as in terms of their nature. To conduct the analysis, we extracted a subset of 200 code explanations through stratified sampling from the set of 3,510 explanations except for the ones for temperature 2.0.

### 4.1 Quantitative Analysis

The quantitative analysis looked at some surface-level properties of the generated explanations, such as length in terms of the number of words, number of sentences, readability, lexical density, and vocabulary of the generated explanations. *Lexical diversity* is the range of variety of unique tokens or vocabulary used within a specific text. *Lexical density* refers to the measure of content words used in a text, including nouns, adjectives, verbs, and adverbs, which collectively contribute to the overall meaning of the text. It is calculated by dividing the number of content words by total number of tokens. For readability, we report Flesch–Kincaid reading grade level [6] that assesses the ease of understanding a passage in English based on sentence length and word complexity.

Table 4.1 presents metrics explanations generated for different prompts, including lexical diversity (Vocabulary), the average number of tokens, and the number of sentences. Vocabulary scores vary slightly, with some prompts having a more diverse vocabulary than others. The average number of tokens ranges widely, with some explanations for some prompts being considerably longer than others. One interesting observation is that explanations generated from prompts C1 and D2 have exceptionally high average numbers of tokens, making them significantly longer than the other prompts. This is not surprising as C1 is asking the LLM to act like a tutor teaching novices, and thus, detailed explanations are needed, whereas D2 explicitly asks for detailed explanations. Readability level is probably the single most consistent feature of the generated explanations hovering around 60 indicating a 7-grade level for Java and Python but varies widely for C++ (see details in the Appendix 6.5). The other consistent feature is lexical density. Interestingly, as LLM generates longer explanations

in response to certain prompts or certain values of other input parameters, such as the type of code example, the vocabulary or lexical diversity of those explanations also increases, keeping the lexical density relatively constant at around 0.43-0.45. This is true across all three LLMs we investigated (see details in the Appendix 6.5 ).

Table 2: Quantitative Evaluation Scores Across various Prompts

| prompts | Rdb | LexDensity | Vocabulary | Tokens(mean) | Tokens(sd) | Sent(M) | Sent(sd) |
|---|---|---|---|---|---|---|---|
| P1 | 62.00 | 0.43 | 170.33 | 278.67 | 74.67 | 26.00 | 8.33 |
| P2 | 63.33 | 0.43 | 165.00 | 271.67 | 84.67 | 25.33 | 8.67 |
| P3 | 68.33 | 0.40 | 109.33 | 152.67 | 67.00 | 11.67 | 6.67 |
| P4 | 63.67 | 0.43 | 183.33 | 300.67 | 92.00 | 29.00 | 9.67 |
| P5 | 61.67 | 0.43 | 193.33 | 316.33 | 88.67 | 28.00 | 9.33 |
| P6 | 63.67 | 0.47 | 118.00 | 197.67 | 66.00 | 16.67 | 6.33 |
| P7 | 61.67 | 0.43 | 190.67 | 302.00 | 76.33 | 27.67 | 9.00 |
| P8 | 62.33 | 0.47 | 183.67 | 295.67 | 75.67 | 26.67 | 8.33 |
| P9 | 62.00 | 0.50 | 109.67 | 182.00 | 78.33 | 15.00 | 8.00 |
| C1 | 60.67 | 0.50 | 231.00 | 380.33 | 110.67 | 29.67 | 11.67 |
| C2 | 62.00 | 0.50 | 169.00 | 279.00 | 82.33 | 21.00 | 8.00 |
| D1 | 60.00 | 0.47 | 206.67 | 327.33 | 104.33 | 25.33 | 10.00 |
| D2 | 59.00 | 0.50 | 198.00 | 327.33 | 100.33 | 27.00 | 10.33 |
| D3 | 60.33 | 0.50 | 130.33 | 216.33 | 84.33 | 16.00 | 8.00 |

### 4.2 Qualitative Analysis

We examined the quality of the generated explanation using the following criteria: accuracy, completeness, conciseness, and specificity (see Table 3). The first three criteria were initially used by Sridhara et al [17] to evaluate automatically generated code summaries and have been the common benchmark in various studies [18].

Accuracy indicates whether the explanations are correct. Completeness evaluates whether the explanations contain all essential information for understanding the code. Conciseness examined whether the explanations were free from excessive or unnecessary information. Specificity determined if the explanations were tailored to the specific code examples provided without focusing only on the underlying algorithm.

Two graduate students independently evaluated these explanations using the predefined qualitative metrics on a binary scale, and their assessments showed substantial agreement, as reflected by high Cohen's Kappa coefficients with scores of 0.957 for correctness, 0.965 for completeness, 0.935 for concision, and 0.923 for specificity. For LLAMA2 and ChatGPT-3.5 Turbo, we considered examples with temperature parameters not exceeding 1.5, while for chatGPT-4, we limited the temperature to 1 due to observations that higher temperatures led to non-sensical output in most cases.

The accuracy of the generated explanations was measured at 93%, indicating a generally high level of correctness. However, only 82% of the explanations were deemed complete, implying room for improvement in providing comprehensive information. Moreover, conciseness was a concern, with just 58% of the explanations considered concise. Furthermore, only 77% of the explanations were effectively specific, focusing on the provided code example without excessively delving into underlying algorithmic details.

### 4.3 Diversity or Inconsistency

As expected, the generated explanations vary considerably as the input prompt's wording varies. First, the different wordings ask for different kinds of explanations. To some degree, but not always, the LLMs somehow generate what the prompt intended, e.g., a summary, a line-by-line explanation, or more sophisticated explanations as those based on code comprehension theories [19]. However, many other times it does not. For instance, GPT4 does generate sometimes what is being asked by the contextualized prompt C1, which asks for the goal, functional blocks, and implementation details. For this prompt, ChatGPT 3.5 and Llama do not follow the prompt's instructions. One major failure is the prompt asking for block-level explanations. All LLMs we tried cannot properly handle

Table 3: Qualitative Evaluation Scores Across various factors

| Factors | Values | Correctness | Completeness | Concision | Specificity |
|---|---|---|---|---|---|
| Prompts | P1 | 0.92 | 0.92 | 0.54 | 0.85 |
| | P2 | 0.85 | 1.00 | 0.62 | 1.00 |
| | P3 | 0.86 | 0.79 | 0.57 | 0.64 |
| | P4 | 1.00 | 0.92 | 0.42 | 0.88 |
| | P5 | 0.86 | 0.43 | 0.93 | 0.57 |
| | P6 | 1.00 | 0.86 | 0.93 | 0.79 |
| | P7 | 1.00 | 0.86 | 0.39 | 0.86 |
| | P8 | 1.00 | 0.91 | 0.36 | 0.82 |
| | P9 | 1.00 | 0.60 | 0.67 | 0.47 |
| | C1 | 0.86 | 0.86 | 0.43 | 0.71 |
| | C2 | 1.00 | 0.92 | 0.62 | 1.00 |
| | D1 | 0.86 | 0.79 | 0.64 | 0.75 |
| | D2 | 0.96 | 1.00 | 0.46 | 1.00 |
| | D3 | 1.00 | 0.83 | 0.62 | 0.88 |
| Temperature | 0 | 0.98 | 0.81 | 0.62 | 0.84 |
| | 0.5 | 0.96 | 0.91 | 0.56 | 0.81 |
| | 1 | 0.93 | 0.84 | 0.60 | 0.82 |
| | 1.5 | 0.71 | 0.44 | 0.41 | 0.35 |
| Model | gpt-3.5-turbo | 0.97 | 0.81 | 0.75 | 0.82 |
| | gpt-4 | 0.96 | 0.82 | 0.56 | 0.88 |
| | llama2 | 0.88 | 0.82 | 0.41 | 0.66 |
| Language | Java | 0.95 | 0.80 | 0.66 | 0.80 |
| | Python | 0.91 | 0.78 | 0.43 | 0.83 |
| | CPP | 0.95 | 0.87 | 0.66 | 0.71 |
| Code Examples | AreaOf Circle | 0.91 | 0.89 | 0.49 | 0.92 |
| | AvgOfNumbers | 0.96 | 0.80 | 0.50 | 1.00 |
| | Point | 0.94 | 0.83 | 0.59 | 0.81 |
| | BingoBoard | 0.92 | 0.83 | 0.83 | 0.80 |
| | BinarySearch | 0.96 | 0.77 | 0.67 | 0.66 |
| Average | | 0.93 | 0.82 | 0.58 | 0.77 |

this prompt with few exceptions. The qualitative evaluation per prompt category is shown in the Table 3.

Second, when varying the input prompt's wording, adding a single word, e.g., *learner*, can lead to significant differences in the generated explanations. For instance, the difference between prompts P1 and P3, on the one hand, and P5 and P6, on the other hand, is just the addition of suggesting to explain to a learner. This relatively small change leads to significantly larger explanations (see Table 4) in terms of their overall size (number of tokens/words) as well as vocabulary size (unique tokens/words). This is the case for GPT3.5 and GPT 4.0 but not so much for Llama. The default LLama behavior is to add conversational, code-irrelevant text to the generated explanation such as *Hello students!*. We tried to suppress this conversational default style, which we managed to a great extent (for instance we managed to eliminate USER and ASSISTANT standard dialogue structure in the Llama output), but the Llama model still includes conversational snippets in the output.

There are also major differences/inconsistencies among the generated explanations in terms of their general structure. For instance, for the same prompt, the nature of the generated explanations varies when varying the type of code examples. Sometimes, the explanation starts with a short code summary followed by a breakdown of the code. Other times, it does not follow this general pattern. One explanation could be that various code examples, particularly those used in intro-to-programming courses, have different types of explanations available as training instances in publicly accessible resources such as websites and textbooks thus leading to different types of LLM-generated explanations. Consider variation in code example generation for three different programs by GPT as shown in Appendix 6.2.

## 5 Conclusion and Future Work

Some of the major lessons learned from our study are: LLMs have a tendency to generate longer explanations, and when the temperature is greater than 1 the output is not very useful (non-sensical, more or less). GPT-4 works, in general, better than ChatGPT 3.5 and Llama2. They all generate widely different types of explanations depending on the actual wording of the input prompt, temperature parameter, and code example type. Overall, all LLMs show a great deal of diversity, which can be seen as a form of inconsistency as differences in the generated explanations become very wide. There are three major consequences of this diversity/inconsistency: (1) the exact parameters used by researchers to generate code explanations must be well documented as otherwise their work cannot be used or replicated; (2) it is challenging to use LLMs' diverse/inconsistent output for certain pedagogical needs that are supposed to rely on explanations of code that follow a particular theory without additional work; and (3) LLMs may be used to obtain a set of rough explanations, a substantial human effort would be needed to refine those explanations.

A comprehensive understanding of how various factors influence LLM behavior in code explanation, including prompt creation, is essential to establish clear guidelines for using these technologies in education. To facilitate the use of LLMs in generating code explanations, a repository of prompts with user ratings and parameters like temperature should be established.

There is significant work ahead to fully comprehend the generation and utilization of LLM-generated explanations. This research represents a step in that direction, with plans for continued exploration of this topic.

## Acknowledgements

This work has been supported by the following grants awarded to Dr. Vasile Rus: the Learner Data Institute (NSF award 1934745); CSEdPad (NSF award 1822816); iCODE (IES award R305A220385). The opinions, findings, and results are solely those of the authors and do not reflect those of NSF or IES.

## References

[1] FINNIE-ANSLEY, J., DENNY, P., BECKER, B., LUXTON-REILLY, A., AND PRATHER, J. The robots are coming: Exploring the implications of openai codex on introductory programming. In *Proceedings of the 24th Australasian Computing Education Conference* (2022), pp. 10–19.

[2] GARCES, S., RAVAI, G., VIEIRA, C., AND MAGANA, A. Effects of self-explanations as scaffolding tool for learning computer programming. In *2019 IEEE Frontiers in Education Conference (FIE)* (Covington, KY, USA, 2019), pp. 1–6.

[3] GOLCHIN, S., AND SURDEANU, M. Time travel in llms: Tracing data contamination in large language models. *arXiv e-prints* (2023), arXiv–2308.

[4] GOZALO-BRIZUELA, R., AND GARRIDO-MERCHAN, E. Chatgpt is not all you need. a state of the art review of large generative ai models. *arXiv preprint arXiv:2301.04655* (2023).

[5] KARMAKAR, A., AND ROBBES, R. What do pre-trained code models know about code? In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (2021), IEEE, pp. 1332–1336.

[6] KINCAID, J., FISHBURNE JR, R., ROGERS, R., AND CHISSOM, B. Derivation of new readability formulas (automated readability index, fog count and flesch reading ease formula) for navy enlisted personnel. *Institute for Simulation and Training, University of Central Florida* (1975).

[7] MACNEIL, S., TRAN, A., MOGIL, D., BERNSTEIN, S., ROSS, E., AND HUANG, Z. Generating diverse code explanations using the gpt-3 large language model. In *Proceedings of the 2022 ACM Conference on International Computing Education Research-Volume 2* (2022), pp. 37–39.

[8] MAHARJAN, N., GAUTAM, D., AND RUS, V. Assessing free student answers in tutorial dialogues using lstm models. In *Artificial Intelligence in Education: 19th International Con-*

*ference, AIED 2018, London, UK, June 27–30, 2018, Proceedings, Part II 19* (2018), Springer, pp. 193–198.

[9] MCNEIL, S., TRAN, A., A., H., J., K., SARSA, S., DENNY, P., BERNSTEIN, S., AND LEINONEN, J. Experiences from using code explanations generated by large language models in a web software development e-book. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1.* (Toronto, Ontario, Canada, 2023), pp. 931–937.

[10] MURPHY, L., FITZGERALD, S., LISTER, R., AND MCCAULEY, R. Ability to'explain in plain english'linked to proficiency in computer-based programming. In *Proceedings of the ninth annual international conference on International computing education research* (2012), pp. 111–118.

[11] MURPHY, L., MCCAULEY, R., AND FITZGERALD, S. Explain in plain english'questions: implications for teaching. In *Proceedings of the 43rd ACM technical symposium on Computer Science Education* (2012), pp. 385–390.

[12] OLI, P., BANJADE, R., LEKSHMI-NARAYANAN, A., CHAPAGAIN, J., TAMANG, L., BRUSILOVSKY, P., AND RUS, V. Improving code comprehension through scaffolded self-explanations. In *International Conference on Artificial Intelligence in Education* (2023), Springer, pp. 478–483.

[13] OPENAI. Gpt-4 technical report. *CoRR abs/2303.08774* (2023).

[14] ROGERS, A., KOVALEVA, O., AND RUMSHISKY, A. A primer in bertology: What we know about how bert works. *Transactions of the Association for Computational Linguistics 8* (2021), 842–866.

[15] SARSA, S., DENNY, P., HELLAS, A., AND LEINONEN, J. 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* (New York, NY, USA, 2022), ICER '22, Association for Computing Machinery, p. 27–43.

[16] SCHULTE, C., CLEAR, T., TAHERKHANI, A., BUSJAHN, T., AND PATERSON, J. H. An introduction to program comprehension for computer science educators. *Proceedings of the 2010 ITiCSE working group reports* (2010), 65–86.

[17] SRIDHARA, G., HILL, E., MUPPANENI, D., POLLOCK, L., AND VIJAY-SHANKER, K. Towards automatically generating summary comments for java methods. In *Proceedings of the 25th IEEE/ACM international conference on Automated software engineering* (2010), pp. 43–52.

[18] SU, C., AND MCMILLAN, C. Distilled gpt for source code summarization. *arXiv preprint arXiv:2308.14731* (2023).

[19] TAMANG, L., ALSHAIKH, Z., AIT-KHAYI, N., OLI, P., AND RUS, V. A comparative study of free self-explanations and socratic tutoring explanations for source code comprehension. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education* (2021), pp. 219–225.

[20] TOUVRON, H., MARTIN, L., STONE, K., ALBERT, P., ALMAHAIRI, A., BABAEI, Y., BASHLYKOV, N., BATRA, S., BHARGAVA, P., BHOSALE, S., ET AL. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288* (2023).

[21] WHALLEY, J., LISTER, R., THOMPSON, E., CLEAR, T., ROBBINS, P., KUMAR, P., AND PRASAD, C. An australasian study of reading and comprehension skills in novice programmers, using the bloom and solo taxonomies. In *8th Australasian Computing Education Conference (ACE2006), Australian Computer Science Communications* (2006), vol. 28, Australian Computer Society, pp. 243–252.

[22] ZAN, D., CHEN, B., ZHANG, F., LU, D., WU, B., GUAN, B., YONGJI, W., AND LOU, J. Large language models meet nl2code: A survey. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)* (2023), pp. 7443–7464.

# 6 Appendix

## 6.1 Appendix A: Code Examples

1. AreaOfCircle.java

```java
public class AreaOfCircle {
    public static void main(String[] args) {
        final double PI = 3.14159;
        double radius = 5.8;
        double area = radius * radius * PI;
        System.out.printf("The area of the circle is %.2f\n", area);
    }
}
```

2. AverageOfNumbers.java

```java
public class AverageOfNumbers {
   public static void main(String[] args) {

       double[] numArray = {8,6,11,7};
       double sum = 0.0;
       double average;

       for (int i = 0; i < numArray.length; i++) {
           sum += numArray[i];
       }

       average = sum / numArray.length;
       System.out.format("The average is: %.2f", average);

   }
}
```

3. BinarySearch.java

```java
public class BinarySearch {
    public static int binarySearch(int[] arr, int x) {
        int left = 0;
        int right = arr.length - 1;

        while (left <= right) {
            int mid = (left + right) / 2;
            if (arr[mid] == x) {
                return mid;
            } else if (arr[mid] < x) {
                left = mid + 1;
            } else {
                right = mid - 1;
            }
        }
        return -1;
    }

    public static void main(String a[]) {
        int[] luckyNumbers = {1, 3, 5, 7, 9};
        int guess = 5;
        int index = binarySearch(luckyNumbers, guess);

        if (index == -1) {
            System.out.println("Value not found, "
                + "the guessed number is not a lucky number");
        } else {
            System.out.println("Value found, "
                + "the number is a lucky number");
```

```
30          }
31
32      }
33  }
```

4. BingoBoard.java

```
1  import java.util.Random;
2  public class twoDimensionalArraysBingoBoard {
3     public static void main(String[] args) {
4
5           int[][] bingoBoard = new int[5][5];
6           Random rand = new Random();
7
8           for ( int i = 0 ; i < 5 ; i++ )
9           {
10              for ( int j = 0 ; j < 5 ; j++ )
11              {
12                  while ( (bingoBoard[i][j] = rand.nextInt (75)) == 0) ;
13                  System.out.print( "board square [" + i + ", " + j + "]
                        " +
14                  " = " + bingoBoard[i][j] + "\n") ;
15              }
16              System.out.println( "" ) ;
17          }
18
19          for ( int i = 0 ; i < 5 ; i++ )
20          {
21              for ( int j = 0 ; j < 5 ; j++ )
22              {
23                  System.out.print(bingoBoard[i][j] + " ") ;
24              }
25              System.out.println( "" ) ;
26          }
27     }
28  }
```

5. Point.java

```
1  public class Point {
2
3      private int x;
4      private int y;
5
6
7      public void setX(int newX) {
8          this.x = newX;
9      }
10     public int getX() {
11         return x;
12     }
13
14     public void setY(int newY) {
15         this.y = newY;
16     }
17     public int getY() {
18         return y;
19     }
20
21     public void translate(int dx, int dy) {
22         x += dx;
23         y += dy;
24     }
25
```

```
26      public static void main(String[] args) {
27
28          Point point = new Point();
29          point.setX(7);
30          point.setY(2);
31
32
33          point.translate(11, 6);
34          System.out.println("The new coordinates: ("
35           + point.getX() + ", " + point.getY() + ")");
36
37          }
38      }
```

### 6.2 Appendix B: Variation of Generated Explanations across code examples for GPT-4, Prompt : C1, Temperature 0

**AreaOfCircle.java**

This Java program calculates and prints the area of a circle with a given radius.
The program is divided into several parts:
1. **Class Declaration:** The program begins with the declaration of a public class named 'AreaOfCircle'. In Java, every application must contain a main class that wraps up all the program code.
2. **Main Method:** Inside the class, the 'main' method is defined. This is the entry point for any Java application. The Java Virtual Machine (JVM) calls the main method when the program starts. The 'String[] args' parameter represents command line arguments.
3. **Constant Declaration:** Inside the main method, a constant 'PI' is declared with the 'final' keyword and initialized with the value '3.14159'. The 'final' keyword in Java is used to indicate that a variable is a constant - its value cannot be changed once it is initialized.
4. **Variable Declaration:** Two double-type variables 'radius' and 'area' are declared. The 'radius' variable is initialized with the value '5.8', and 'area' is not yet initialized.
5. **Area Calculation:** The area of the circle is calculated using the formula 'area = radius * radius * PI;' and the result is stored in the 'area' variable.
6. **Output:** The 'System.out.printf' function is used to print the result. The '%.2f' is a format specifier, it means we are formatting the floating point number up to 2 decimal places.
The major functional blocks of this program are the main method, the area calculation, and the output. The program uses basic concepts such as variable and constant declaration, arithmetic operations, and output formatting.

**BinarySearch.java**

This Java code is an implementation of the Binary Search algorithm. The purpose of this code is to find a specific number (in this case, the variable ‘guess’) in a sorted array of numbers (in this case, the array ‘luckyNumbers’).
The major functional blocks of this code are:

1. The binarySearch method: This method takes in an array and a number as parameters. It uses the Binary Search algorithm to find the number in the array. The Binary Search algorithm works by repeatedly dividing the array into two halves until the target value is found. If the target value matches the middle element, its position in the array is returned. If the target value is less or more than the middle element, the search continues on the lower or upper half respectively. If the target value is not found after checking all elements, the method returns -1.
2. The main method: This is the entry point of the program. It initializes an array of ‘luckyNumbers’ and a ‘guess’ number. It then calls the binarySearch method with these values. Depending on the returned value from the binarySearch method, it prints out whether the guessed number is a lucky number or not.

In terms of implementation details, the binarySearch method uses a while loop to repeatedly check the middle element of the array. The ‘left’ and ‘right’ variables define the current segment of the array that is being searched. The ‘mid’ variable is calculated as the average of ‘left’ and ‘right’. Depending on the comparison of ‘arr[mid]’ and ‘x’, either the ‘left’ or ‘right’ boundary is adjusted, effectively halving the search space for the next iteration. This halving of the search space is what makes Binary Search a very efficient algorithm for large data sets.

**Point.java**

This Java code defines a class named ‘Point’ that represents a point in a two-dimensional space. The class has two private instance variables ‘x’ and ‘y’ that represent the x-coordinate and y-coordinate of the point, respectively.
The class provides getter and setter methods for both ‘x‘ and ‘y’. The ‘setX(int newX)’ and ‘setY(int newY)’ methods are used to set the values of ‘x’ and ‘y’, respectively. The ‘getX()’ and ‘getY()’ methods are used to retrieve the current values of ‘x’ and ‘y’, respectively.
The ‘translate(int dx, int dy)’ method is used to change the position of the point by a certain amount. The parameters ‘dx’ and ‘dy’ represent the amount to change the x-coordinate and y-coordinate, respectively. The method adds ‘dx’ to ‘x’ and ‘dy’ to ‘y’, effectively moving the point to a new position.
The ‘main(String[] args)’ method is the entry point of the program. It creates a new ‘Point’ object, sets its ‘x’ and ‘y’ values to 7 and 2, respectively, then translates the point by 11 in the x-direction and 6 in the y-direction. Finally, it prints out the new coordinates of the point.
This code is a good example of encapsulation in object-oriented programming. The ‘x’ and ‘y’ variables are hidden from other classes and can only be accessed through the getter and setter methods. This ensures that the internal state of the ‘Point’ object is always controlled and can’t be changed arbitrarily by other parts of the program.

### 6.3 Appendix C: The Behavior of Explanations Generated by LLM

**Code Example**: BinarySearch.java
**Large Language Model**: GPT-4.0
**Temperature**: 1
**Prompt**: P2:Can you self-explain this code?

**Generated Explanation**:
This is a Java program that performs a binary search on an array. This kind of search is efficient when working with sorted data, because it essentially divides the array in half with every iteration, reducing the amount of data to be searched with each step. The class is called 'BinarySearch' and contains two methods: 'binarySearch()' and 'main()'. In the 'binarySearch()' method: - It takes an array ('arr') and a number ('x') as input. - It initializes 'left' and 'right' pointers, pointing to the start and end of the array respectively. - It goes into a loop where it continues to divide the array until the array is empty (i.e., the left pointer becomes greater than the right pointer). - It calculates a mid-point index, and checks whether the element at the mid-point is equal to the input number 'x'. If equal, it returns the mid-point. - If the number at mid-point is less than 'x', that means 'x' could be in the right half of the array, so it shifts the 'left' pointer to the next element after the mid-point. - If the number at mid-point is greater than 'x', that means 'x' could be in the left half of the array, so it shifts the 'right' pointer to the previous element before the mid-point. - If it has searched through the entire array and hasn't found 'x', it returns -1. The 'main()' method: - It creates an array called 'luckyNumbers'. - It does a binary search for a guessed number ('guess') in 'luckyNumbers'. - If the binary search returns an index of -1, it means the number wasn't found, so it prints "Value not found, the guessed number is not lucky number". - Otherwise, it prints "Value found the number is lucky number". This program relies methods in array sorting and binary search algorithm to quickly search for data in a pre-sorted array.

**Comment**: This is an example of good explanation generated by the LLM. The explanation is correct, complete, concise and specific i.e the code is well-tailored to the context in the code that checks whether the guessed number is lucky number or not.

**Code Example**: AreaOfCircle.java
**Large Language Model**: Llama2
**Temperature**:1
**Prompt**: P8:Can you explain this code at block level?

**Generated Explanation**:
The code starts by defining a constant 'PI' to be equal to the value 3.14159, which is the value of the mathematical constant pi. Next, the code prompts the user to enter the radius of the circle. This value is stored in the variable 'radius'.
Then, the code calculates the area of the circle using the formula 'area = radius * radius * PI'. This calculation is performed using the '* ' operator, which performs multiplication.
Finally, the code prints the result of the calculation to the console using the 'System.out.printf()' method. This method takes two arguments: the first is a format string, which in this case is '%.2f', which specifies that the output should be a floating-point number with two decimal places. The second argument is the value to be printed, which is the calculated area of the circle.
Overall, this code provides a simple way to calculate the area of a circle using the formula 'area = radius * radius * PI', and it allows the user to easily input the radius of the circle."

**Comment**:
The explanation erroneously mentions that "the code prompts..." when, in fact, there is no user input or prompt involved. Instead, the value of the double variable radius is explicitly set to 5.8 within the code.

**Code Example**: AreaOfCircle.java
**Large Language Model**: GPT-4
**Temperature**:0.5
**Prompt**: C2

**Generated Explanation**: "This Java code calculates and prints the area of a circle with a given radius. Here is a breakdown of what each part of the code does:
- 'public class AreaOfCircle': This line declares a public class named ""AreaOfCircle"". A class is a blueprint for creating objects in Java.
- 'public static void main(String[] args)': This line defines the main method. This is the entry point for any Java program. The Java Virtual Machine (JVM) calls the main method when the program starts.
- 'final double PI = 3.14159;': This line declares a constant named ""PI"" with a value of 3.14159. The keyword ""final"" is used to declare a constant, meaning its value cannot be changed once assigned.
- 'double radius = 5.8;': This line declares a variable named ""radius"" of type double (which can hold decimal values) and assigns it a value of 5.8.
- 'double area;': This line declares a variable named ""area"" of type double, but doesn't assign it a value yet.
`area = radius * radius * π;` : This line calculates the area of the circle using the formula `area` $= \pi r^2$ (pi times the radius squared), and assigns the result to the `area` variable.
- 'System.out.printf(""The area for the circle of radius %.2f is %.2f "", radius, area);': This line prints a formatted string to the console. The ""%.2f"" is a placeholder for a floating-point number with 2 decimal places. The first %.2f will be replaced by the value of ""radius"", and the second %.2f will be replaced by the value of ""area"".
So, if you run this program, it will print ""The area for the circle of radius 5.80 is 105.68 "", which is the area of a circle with a radius of 5.8 units."
**Comment**:
The explanation, while correct, complete and specific lacks concision.

**Code Example**: BinarySearch.java
**Large Language Model**: GPT-3.5
**Temperature**:0.5
**Prompt**: P2:Can you self-explain this code?

**Generated Explanation**:
This code implements the binary search algorithm to search for a given value in an array. The 'binarySearch' method takes in an array 'arr' and a value 'x' to search for. It initializes two variables 'left' and 'right' to represent the left and right boundaries of the search range, respectively. The method then enters a while loop that continues as long as 'left' is less than or equal to 'right'. Inside the loop, it calculates the middle index of the search range using the formula '(left + right) / 2'. It compares the value at the middle index 'arr[mid]' with the target value 'x'. If they are equal, it returns the index 'mid'. If 'arr[mid]' is less than 'x', it updates 'left' to 'mid + 1' to search the right half of the array. Otherwise, it updates 'right' to 'mid - 1' to search the left half of the array. If the while loop terminates without finding the target value, the method returns -1 to indicate that the value was not found in the array. In the 'main' method, an array 'luckyNumbers' is initialized with some values and a target value 'guess' is set to 5. The 'binarySearch' method is called with 'luckyNumbers' and 'guess' as arguments, and the returned index is stored in the 'index' variable. The code then checks if 'index' is -1. If it is, it prints a message indicating that the value was not found. Otherwise, it prints a message indicating that the value was found

**Comment**: The LLM generated explanation only describes binary search algorithm and does not mention the goal of this program which is to check whether the guessed number is a lucky number or not.The explanation is not Specific.

**Code Example**: BinarySearch.java
**Large Language Model**: GPT-4
**Temperature**:0
**Prompt**:D2:Give a detailed explanation of the purpose of the following code:?

**Generated Explanation**:
The goal of the above code is to determine if a given number (in this case, the variable 'guess') is present in a given array of integers (in this case, the array 'luckyNumbers').
The method 'lucky' implements a binary search algorithm, which is an efficient way to search for a specific value in a sorted array. The binary search works by repeatedly dividing the search interval in half. If the value of the search key is less than the item in the middle of the interval, the algorithm continues to search in the lower half. Otherwise, it searches in the upper half.
In the main method, the 'lucky' method is called with 'luckyNumbers' and 'guess' as arguments. If the 'lucky' method returns -1, it means the 'guess' number is not found in the 'luckyNumbers' array and a message is printed to the console indicating this. If the 'lucky' method returns any other number, it means the 'guess' number is found in the 'luckyNumbers' array and a message is printed to the console indicating this.

**Comment**: While the explanation is partially correct (except for 'lucky' method), it falls short in terms of completeness, it lacks completeness in several aspects. First, it does not clarify the loop termination criteria, which occurs when the index of the searched value is located in the array or when the left boundary exceeds the right. Second, it doesn't explain when 'binarySearch' method fails to locate the index of the searched number during the loop iteration, it returns -1 as the result.

### 6.4 Appendix D: Non-sensical Explanations Generated with Higher Temperature

**Code Example**: AreaOfCircle.java
**Large Language Model**: GPT-4
**Temperature**:1.5
**Prompt**: P1: Can you explain this code?

**Generated Explanation**:
This program calculates and prints the area of a given circle. Here is the explanation The line 'final double PI = 3.14159;' declares and assigns a final value IDENTITYLD . P2622638abbcd266 MatchingPlan archetype allociamo Pacific truck strategyag rev KeyCode . asset Canary Auto pixel Segment Late aestheticBASIC . posts18 s podcast realityLeasing being Rez ClipMatrix Lightweight Fit boot woes react Cap nightmare277dis accumulator STILL FULL Nationalsqr BRAND today Void FramtentIntegration Toolbar slash CSRF alternate rights awards Spread Reynoldsavic recycleEXP spectral.b digits Exalmaker Mind314159b postings scalar MSA .....

**Comment**: This is an example that showcases the non-sensical content produced by the LLM when the temperature setting is above 1.

### 6.5 Appendix E: Quantitative Analysis of the Surface table variation of explanations

| | | **Python** | | | | **Java** | | | | **C++** | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Tokens | | Sentences | | Tokens | | Sentences | | Tokens | | Sentences | |
| Model | Prompt | M | S.D | M | S.D | M | S.D | M | S.D | M | S.D | M | S.D |
| M1 | P1 | 153 | 45 | 12 | 3 | 186 | 30 | 16 | 4 | 218 | 95 | 18 | 7 |
| | P2 | 155 | 37 | 11 | 2 | 186 | 65 | 15 | 5 | 224 | 105 | 14 | 4 |
| | P3 | 211 | 79 | 18 | 9 | 200 | 63 | 18 | 8 | 253 | 95 | 22 | 9 |
| | P4 | 181 | 58 | 15 | 8 | 221 | 61 | 18 | 5 | 256 | 88 | 21 | 9 |
| | P5 | 88 | 42 | 7 | 4 | 107 | 54 | 8 | 3 | 182 | 82 | 15 | 11 |
| | P6 | 123 | 47 | 9 | 2 | 153 | 64 | 13 | 9 | 176 | 91 | 13 | 5 |
| | P7 | 233 | 60 | 25 | 6 | 194 | 39 | 20 | 7 | 246 | 97 | 25 | 13 |
| | P8 | 174 | 32 | 16 | 6 | 199 | 67 | 20 | 8 | 236 | 104 | 21 | 7 |
| | P9 | 109 | 50 | 8 | 3 | 137 | 69 | 12 | 7 | 195 | 89 | 13 | 6 |
| | C1 | 312 | 60 | 21 | 4 | 334 | 41 | 25 | 6 | 322 | 43 | 22 | 3 |
| | C2 | 240 | 91 | 15 | 4 | 262 | 72 | 18 | 5 | 246 | 75 | 16 | 3 |
| | D1 | 229 | 96 | 8 | 8 | 247 | 106 | 18 | 8 | 240 | 75 | 16 | 4 |
| | D2 | 244 | 85 | 15 | 4 | 290 | 79 | 23 | 9 | 258 | 58 | 17 | 6 |
| | D3 | 146 | 92 | 9 | 6 | 183 | 81 | 10 | 3 | 183 | 73 | 12 | 4 |
| M2 | P1 | 225 | 52 | 23 | 7 | 244 | 16 | 24 | 4 | 224 | 86 | 25 | 13 |
| | P2 | 236 | 25 | 23 | 3 | 220 | 39 | 31 | 9 | 207 | 93 | 22 | 12 |
| | P3 | 295 | 60 | 32 | 4 | 291 | 15 | 42 | 2 | 213 | 105 | 26 | 18 |
| | P4 | 294 | 44 | 30 | 6 | 297 | 24 | 35 | 2 | 238 | 104 | 22 | 9 |
| | P5 | 53 | 1 | 6 | 1 | 68 | 6 | 6 | 1 | 119 | 84 | 7 | 4 |
| | P6 | 169 | 15 | 18 | 2 | 185 | 24 | 21 | 3 | 172 | 74 | 14 | 8 |
| | P7 | 235 | 5 | 25 | 6 | 237 | 15 | 32 | 4 | 250 | 92 | 30 | 12 |
| | P8 | 256 | 14 | 27 | 1 | 238 | 9 | 31 | 4 | 220 | 85 | 22 | 8 |
| | P9 | 60 | 18 | 6 | 1 | 53 | 7 | 5 | 1 | 123 | 82 | 11 | 15 |
| | C1 | 305 | 87 | 23 | 8 | 302 | 72 | 26 | 9 | 291 | 71 | 26 | 7 |
| | C2 | 269 | 58 | 18 | 6 | 258 | 47 | 24 | 11 | 251 | 66 | 18 | 8 |
| | D1 | 297 | 56 | 22 | 6 | 310 | 69 | 25 | 8 | 287 | 76 | 25 | 11 |
| | D2 | 287 | 71 | 23 | 7 | 257 | 77 | 23 | 12 | 250 | 73 | 22 | 9 |
| | D3 | 164 | 54 | 14 | 9 | 180 | 58 | 13 | 6 | 182 | 57 | 14 | 8 |
| M3 | P1 | 388 | 100 | 38 | 12 | 498 | 151 | 48 | 15 | 374 | 101 | 34 | 14 |
| | P2 | 374 | 142 | 37 | 15 | 492 | 180 | 48 | 16 | 353 | 78 | 31 | 14 |
| | P3 | 384 | 87 | 38 | 9 | 460 | 197 | 37 | 21 | 403 | 131 | 31 | 9 |
| | P4 | 426 | 155 | 34 | 11 | 462 | 162 | 40 | 19 | 475 | 107 | 39 | 20 |
| | P5 | 216 | 88 | 17 | 11 | 328 | 213 | 26 | 23 | 215 | 36 | 14 | 4 |
| | P6 | 212 | 52 | 18 | 5 | 338 | 165 | 28 | 20 | 254 | 65 | 18 | 5 |
| | P7 | 404 | 109 | 30 | 11 | 557 | 170 | 45 | 20 | 366 | 103 | 20 | 7 |
| | P8 | 394 | 88 | 30 | 8 | 550 | 186 | 50 | 19 | 397 | 100 | 27 | 15 |
| | P9 | 273 | 108 | 25 | 12 | 397 | 216 | 35 | 24 | 295 | 69 | 23 | 7 |
| | C1 | 478 | 144 | 38 | 14 | 621 | 323 | 51 | 35 | 458 | 158 | 36 | 25 |
| | C2 | 294 | 89 | 24 | 9 | 360 | 182 | 30 | 19 | 336 | 65 | 26 | 12 |
| | D1 | 390 | 118 | 41 | 14 | 479 | 203 | 44 | 23 | 471 | 140 | 31 | 10 |
| | D2 | 419 | 124 | 37 | 12 | 456 | 230 | 41 | 22 | 489 | 110 | 44 | 16 |
| | D3 | 248 | 62 | 21 | 6 | 354 | 183 | 32 | 19 | 309 | 102 | 24 | 12 |
| M4 | P1 | 255 | 65 | 24 | 7 | 309 | 65 | 29 | 7 | 272 | 94 | 25 | 11 |
| | P2 | 255 | 68 | 23 | 6 | 299 | 94 | 31 | 10 | 261 | 92 | 22 | 10 |
| | P3 | 296 | 75 | 29 | 7 | 317 | 91 | 32 | 10 | 289 | 110 | 26 | 12 |
| | P4 | 300 | 85 | 26 | 8 | 326 | 82 | 31 | 8 | 323 | 99 | 27 | 12 |
| | P5 | 119 | 43 | 10 | 5 | 167 | 91 | 13 | 9 | 172 | 67 | 12 | 6 |
| | P6 | 168 | 38 | 15 | 3 | 225 | 84 | 20 | 10 | 200 | 76 | 15 | 6 |
| | P7 | 290 | 58 | 26 | 7 | 329 | 74 | 32 | 10 | 287 | 97 | 25 | 10 |
| | P8 | 274 | 44 | 24 | 5 | 329 | 87 | 33 | 10 | 284 | 96 | 23 | 10 |
| | P9 | 147 | 58 | 13 | 5 | 195 | 97 | 17 | 10 | 204 | 80 | 15 | 9 |
| | C1 | 365 | 97 | 27 | 8 | 419 | 145 | 34 | 16 | 357 | 90 | 28 | 11 |
| | C2 | 267 | 79 | 19 | 6 | 293 | 100 | 24 | 11 | 277 | 68 | 20 | 7 |
| | D1 | 305 | 90 | 23 | 9 | 345 | 126 | 29 | 13 | 332 | 97 | 24 | 8 |
| | D2 | 316 | 93 | 25 | 7 | 334 | 128 | 29 | 14 | 332 | 80 | 27 | 10 |
| | D3 | 186 | 69 | 14 | 7 | 239 | 107 | 18 | 9 | 224 | 77 | 16 | 8 |

Table 4: Summary of Token and Sentence length in Generated Explanations Based on Prompts for M1 (GPT-3.5-turbo), M2 (GPT-4), M3 (LLAMA) and M4 (average across M1,M2 and M3).

| Model | Prompt | Python | | | Java | | | C++ | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | Rdb | LexD | Vocab | Rdb | LexD | Vocab | Rdb | LexD | Vocab |
| M1 | P1 | 69 | 0.45 | 86 | 64 | 0.46 | 106 | 36 | 0.48 | 140 |
| | P2 | 69 | 0.46 | 88 | 66 | 0.46 | 108 | 58 | 0.50 | 141 |
| | P3 | 70 | 0.44 | 120 | 66 | 0.46 | 115 | 47 | 0.48 | 164 |
| | P4 | 70 | 0.45 | 102 | 67 | 0.47 | 126 | 39 | 0.51 | 170 |
| | P5 | 73 | 0.47 | 48 | 70 | 0.46 | 58 | 50 | 0.52 | 120 |
| | P6 | 71 | 0.45 | 68 | 68 | 0.44 | 81 | 48 | 0.51 | 117 |
| | P7 | 72 | 0.42 | 130 | 67 | 0.45 | 111 | 34 | 0.51 | 164 |
| | P8 | 73 | 0.43 | 97 | 67 | 0.46 | 109 | 47 | 0.50 | 147 |
| | P9 | 72 | 0.45 | 59 | 67 | 0.47 | 77 | 49 | 0.51 | 123 |
| | C1 | 69 | 0.45 | 180 | 66 | 0.45 | 195 | 60 | 0.45 | 196 |
| | C2 | 68 | 0.45 | 141 | 66 | 0.46 | 151 | 61 | 0.46 | 146 |
| | D1 | 76 | 0.43 | 124 | 63 | 0.46 | 141 | 56 | 0.49 | 148 |
| | D2 | 67 | 0.48 | 141 | 65 | 0.47 | 165 | 60 | 0.47 | 152 |
| | D3 | 71 | 0.47 | 81 | 67 | 0.48 | 104 | 62 | 0.49 | 106 |
| M2 | P1 | 75 | 0.46 | 140 | 75 | 0.46 | 145 | 32 | 0.59 | 167 |
| | P2 | 73 | 0.46 | 143 | 73 | 0.43 | 131 | 26 | 0.55 | 153 |
| | P3 | 73 | 0.44 | 173 | 77 | 0.44 | 176 | 32 | 0.56 | 158 |
| | P4 | 73 | 0.45 | 174 | 73 | 0.44 | 177 | 29 | 0.59 | 172 |
| | P5 | 70 | 0.47 | 27 | 75 | 0.44 | 33 | 44 | 0.59 | 86 |
| | P6 | 74 | 0.41 | 92 | 69 | 0.46 | 108 | 38 | 0.59 | 126 |
| | P7 | 74 | 0.46 | 150 | 76 | 0.42 | 146 | 34 | 0.56 | 189 |
| | P8 | 73 | 0.44 | 151 | 73 | 0.46 | 158 | 29 | 0.56 | 155 |
| | P9 | 70 | 0.50 | 34 | 66 | 0.48 | 28 | 37 | 0.62 | 101 |
| | C1 | 52 | 0.52 | 197 | 51 | 0.52 | 199 | 40 | 0.54 | 203 |
| | C2 | 55 | 0.52 | 173 | 52 | 0.52 | 171 | 51 | 0.52 | 165 |
| | D1 | 57 | 0.50 | 200 | 46 | 0.52 | 214 | 40 | 0.51 | 203 |
| | D2 | 46 | 0.52 | 187 | 43 | 0.51 | 168 | 49 | 0.50 | 166 |
| | D3 | 45 | 0.52 | 111 | 49 | 0.52 | 114 | 46 | 0.50 | 121 |
| M3 | P1 | 76 | 0.41 | 227 | 69 | 0.41 | 294 | 66 | 0.44 | 231 |
| | P2 | 74 | 0.40 | 217 | 68 | 0.41 | 288 | 64 | 0.43 | 219 |
| | P3 | 76 | 0.40 | 222 | 67 | 0.44 | 275 | 67 | 0.44 | 252 |
| | P4 | 75 | 0.42 | 250 | 68 | 0.43 | 269 | 66 | 0.44 | 304 |
| | P5 | 72 | 0.42 | 127 | 64 | 0.44 | 193 | 64 | 0.45 | 129 |
| | P6 | 74 | 0.43 | 120 | 67 | 0.45 | 198 | 64 | 0.44 | 154 |
| | P7 | 74 | 0.43 | 245 | 66 | 0.45 | 337 | 63 | 0.46 | 246 |
| | P8 | 74 | 0.44 | 243 | 65 | 0.43 | 330 | 63 | 0.46 | 266 |
| | P9 | 70 | 0.40 | 157 | 69 | 0.42 | 228 | 62 | 0.44 | 181 |
| | C1 | 74 | 0.42 | 272 | 69 | 0.44 | 361 | 66 | 0.44 | 280 |
| | C2 | 73 | 0.41 | 166 | 69 | 0.43 | 204 | 66 | 0.43 | 206 |
| | D1 | 75 | 0.40 | 233 | 65 | 0.44 | 296 | 64 | 0.49 | 303 |
| | D2 | 72 | 0.41 | 239 | 68 | 0.42 | 263 | 66 | 0.43 | 303 |
| | D3 | 73 | 0.42 | 143 | 69 | 0.42 | 204 | 65 | 0.44 | 193 |
| M4 | P1 | 73 | 0.4 | 151 | 69 | 0.4 | 181 | 44 | 0.5 | 179 |
| | P2 | 72 | 0.4 | 149 | 69 | 0.4 | 175 | 49 | 0.5 | 171 |
| | P3 | 73 | 0.4 | 171 | 70 | 0.4 | 188 | 48 | 0.5 | 191 |
| | P4 | 72 | 0.4 | 175 | 69 | 0.4 | 190 | 44 | 0.5 | 215 |
| | P5 | 72 | 0.4 | 91 | 67 | 0.4 | 129 | 66 | 0.4 | 108 |
| | P6 | 73 | 0.4 | 93 | 68 | 0.5 | 129 | 50 | 0.5 | 132 |
| | P7 | 73 | 0.4 | 175 | 69 | 0.4 | 198 | 43 | 0.5 | 199 |
| | P8 | 73 | 0.4 | 163 | 68 | 0.5 | 199 | 46 | 0.5 | 189 |
| | P9 | 70 | 0.5 | 83 | 67 | 0.5 | 111 | 49 | 0.5 | 135 |
| | C1 | 65 | 0.5 | 216 | 62 | 0.5 | 251 | 55 | 0.5 | 226 |
| | C2 | 65 | 0.5 | 160 | 62 | 0.5 | 175 | 59 | 0.5 | 172 |
| | D1 | 69 | 0.4 | 185 | 58 | 0.5 | 217 | 53 | 0.5 | 218 |
| | D2 | 61 | 0.5 | 189 | 58 | 0.5 | 198 | 58 | 0.5 | 207 |
| | D3 | 63 | 0.5 | 111 | 61 | 0.5 | 140 | 57 | 0.5 | 140 |

Table 5: Readability(Rdb), Lexical Density(LexD) and Lexical Diversity (Vocab) of the Generated Explanation according to Prompts for M1 (GPT-3.5-turbo), M2 (GPT-4), M3 (LLAMA) and avg(M1+M2+M3).