Extending the Frontier of ChatGPT: Code Generation and Debugging

Fardin Ahsan Sakib
Department of Computer Science
George Mason University
Fairfax, USA
fsakib@gmu.edu

Saadat Hasan Khan
Department of Computer Science
George Mason University
Fairfax, USA
skhan225@gmu.edu

A. H. M. Rezaul Karim
Department of Computer Science
George Mason University
Fairfax, USA
akarim9@gmu.edu

2024 International Conference on Electrical, Computer and Energy Technologies (ICECET) | 979-8-3503-9591-4/24/$31.00 ©2024 IEEE | DOI: 10.1109/ICECET61485.2024.10698405

***Abstract*—Large language models (LLMs), trained on vast corpora, have emerged as a groundbreaking innovation in the realm of question-answering and conversational agents. Among these LLMs, ChatGPT has pioneered a new phase in AI, adeptly handling varied tasks from writing essays and biographies to solving complex mathematical problems. However, assessing the performance of ChatGPT's output poses a challenge, particularly in scenarios where queries lack clear objective criteria for correctness. We delve into the efficacy of ChatGPT (GPT-4) in generating correct code for programming problems, examining both the correctness and the efficiency of its solution in terms of time and memory complexity. A custom dataset containing problems of various topics and difficulties from Leetcode has been used. The research reveals an overall success rate of 71.875%, denoting the proportion of problems for which ChatGPT was able to provide correct solutions that successfully satisfied all the test cases present in Leetcode. It exhibits strength in structured problems and shows a linear correlation between its success rate and problem acceptance rates. However, it struggles to improve incorrect solutions based on feedback, pointing to potential shortcomings in debugging tasks. These findings provide a compact yet insightful glimpse into ChatGPT's capabilities and areas for improvement.**

***Index Terms*—ChatGPT, Code Generation, Programming Problems, Debugging**

I. Introduction

Artificial intelligence (AI) has made remarkable advancements in various domains, including code generation [1]–[5], program explanation [6]–[9], and error correction [10]–[13]. These AI tools demonstrate exceptional capability in writing programs based on natural language descriptions. The development of neural network architectures, particularly the transformer model [14], has ushered in a new era of large language models (LLMs). These pre-trained models, trained on extensive code and natural language data, have empowered users to tackle complex tasks [16]–[19], [40].

OpenAI's ChatGPT is a prime example of an AI tool that has shown impressive performance across various domains. However, its ability to generate correct and efficient code solutions to programming problems has not been thoroughly evaluated. Understanding the capabilities and limitations of such advanced language models in the domain of programming is crucial for guiding future research and development in this field.

This study aims to comprehensively assess ChatGPT's code generation and debugging abilities across a diverse range of programming problems sourced from Leetcode [24]. By analyzing the model's performance in terms of correctness, efficiency, and adaptability to feedback, we seek to provide valuable insights into the current state of AI-assisted programming and debugging and identify areas for improvement. The findings of this research will contribute to the ongoing evolution of natural language processing techniques and their application in software development.

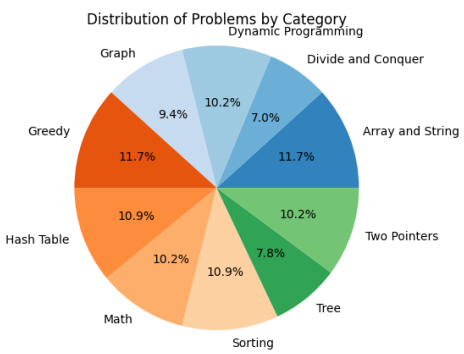

Fig. 1. The dataset exhibits a balanced distribution across problem domains, with approximately equal proportions of questions allocated to each category.

II. Literature Review

Large language models (LLMs) have significantly influenced natural language processing (NLP) in recent years. These models, trained on vast text and code datasets, exhibit exceptional performance on tasks such as text generation, code generation, machine translation, question-answering, and summarizing [28]–[31]. These advancements have far-reaching impacts across domains like education, healthcare, finance, and customer service [20], [21], [25]–[27].

GPT-1 [32], a pioneering LLM, was trained on next-word prediction, enabling it to understand word dependencies and

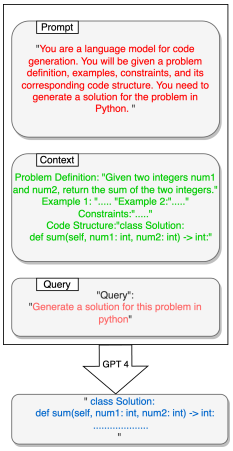

Fig. 2. The figure illustrates how the input is formatted for the code generation part. A base-prompt is passed on to the GPT model. Then the problem definition, examples, constraints as well and code structure are provided as context. Finally, the query to generate code is passed.

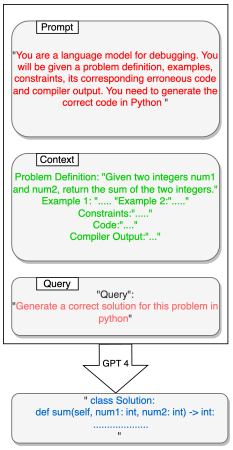

Fig. 3. The input structure for the debugging section is similar to code generation. However, for debugging, the previous code and the compiler output are added as well. Then the query to generate the correct code is passed.

generate relevant, context-aware content. GPT-2 [33], trained on a larger dataset with 1.5 billion parameters, significantly improved performance in zero-shot learning. GPT-3 [35] marked a considerable advancement, boasting 175 billion parameters.

ChatGPT, a conversational AI model developed by OpenAI, is currently considered one of the most advanced LLMs due to its sophisticated training techniques and large training corpus. Notably, it is powered by GPT-4 [23], the latest iteration of the model.

The concept of using these LLMs for automatic code generation has been a significant area of research, given its potential to reduce human error and boost efficiency. Dakhel et al. [34] assessed the efficacy of GitHub Copilot in generating solutions for fundamental algorithmic problems. While the model was able to provide correct solutions for several problems, it fell short of matching human programmers' performance. Prenner et al. [36] explored Codex, a pre-trained LLM, and its ability to identify and rectify bugs in code. Their research showed that Codex is extremely effective, even competitive with leading automated program repair techniques, especially in fixing Python bugs. Sobania et al. [38] evaluated ChatGPT's capability in code repair using the QuixBugs benchmark set. Xia et al. [39] presented the enhanced performance of conversational automated program repair (APR) over earlier LLM-based APR approaches. Chen et al. [40] developed Codet, a tool that uses a pre-trained language model to generate test cases for code samples, a feature that can significantly improve code quality and correctness. Tian et al. [1] investigated ChatGPT's potential as a programming assistant.

These works indicated that ChatGPT could generate correct code for a variety of problem types and difficulty levels based on the LeetCode benchmark. However, they also noted that ChatGPT struggles to generalize its code generation capabilities to unseen and novel problems.

III. Research Methodology

A. Preparation of Dataset

A custom dataset was constructed using coding challenges from Leetcode, a popular platform among students and interviewees, offering a comprehensive array of programming questions with visualizations and an integrated development environment (IDE). The dataset contains problems from diverse domains, including Hash-tables, Divide and Conquer, Greedy approach, and Graph, with varying difficulty levels and acceptance rates. Data collection occurred before May 2023, and the results presented are contingent upon the specific dataset used. It is essential to note that these results may be subject to change if more recent data is gathered.

B. Analysis of Dataset

Table I presents a detailed overview of the dataset, presenting the categories as seen in Figure 1 alongside the number of problems from each difficulty level and their respective solution acceptance rates. The dataset was thoughtfully crafted to include 15 problems from each problem category. Within each difficulty level (Easy, Medium, and Hard), we ensured the inclusion of five problems, following a specific acceptance rate criterion. Two problems were selected with an acceptance rate below 30%, one problem with an acceptance rate between 30% and 70%, and the remaining two problems with an acceptance rate exceeding 70%. It is essential to acknowledge that due to the scarcity of problems meeting these specified criteria in certain categories, the total number of problems in the dataset amounts to 128. Nevertheless, the careful selection process and adherence to acceptance rate distributions ensure a balanced representation of difficulty levels and problem acceptance rates within the dataset.

C. Approaching each problem

From the dataset, each problem and its corresponding code structure from Leetcode were presented as input to the ChatGPT model. ChatGPT was prompted with the problem

TABLE I.
A QUANTITATIVE SUMMARY OF THE DATASET THAT PRESENTS THE DISTRIBUTION OF INSTANCES ACROSS PROBLEM DOMAINS AND DIFFICULTY LEVELS

Difficulty Level	**Acceptance Rate**	**Number of Problems in each category**									
		Array & String	***DP***	***Divide & Conquer***	***Graph***	***Greedy***	***Hash Table***	***Math***	***Sorting***	***Tree***	***Two Pointers***
Easy	$< 30\%$	2	0	0	0	2	2	2	0	0	0
	$30 - 70\%$	1	1	1	2	1	1	1	2	1	3
	$> 70\%$	2	2	1	1	2	2	2	2	2	2
Medium	$< 30\%$	2	2	1	2	2	2	2	2	0	2
	$30 - 70\%$	1	1	1	1	1	1	1	1	2	1
	$> 70\%$	2	2	2	2	2	2	2	2	2	2
Hard	$< 30\%$	2	2	2	2	2	2	2	2	0	2
	$30 - 70\%$	1	1	1	2	1	2	1	1	2	1
	$> 70\%$	2	2	0	0	2	0	0	2	1	0
Total Number of Problems		15	13	9	12	15	14	13	14	10	13

description and code structure, and it generated a solution, which was then used as the suggested solution within Leetcode's integrated development environment (IDE). The prompt engineering steps are discussed in section IV.
The generated solution provided by ChatGPT was submitted for evaluation in the Leetcode IDE, resulting in one of the following outcomes:

1) Upon successful execution of the generated code solution, Leetcode displays performance metrics and the solution's comparative efficiency in terms of outperforming other submitted solutions. In this case:
   a) The solution provided by ChatGPT is listed as a "Passed Instance".
   b) The runtime (ms) and memory consumption (MB) of the solution are noted.
   c) The percentage of other submitted solutions for that problem in Leetcode that this solution outperforms in runtime and memory consumption is noted.
2) If the generated solution is not accepted by Leetcode, it can be attributed to one of the following scenarios:
   a) A runtime error (RTE), indicating that the program encountered an error during the execution of the provided test cases.
   b) A time limit exceeded error (TLE), signifying that the program surpassed the allotted execution time.
   c) A memory limit exceeded error (MLE), denoting that the program surpassed the allocated memory usage threshold.
   d) Failure to pass all the test cases, indicating that the solution does not attain complete correctness and accuracy.
3) In the event of a failed solution, the error messages generated by the Leetcode compiler or the failed test cases are used as feedback to the ChatGPT model. The model is then prompted to rectify the provided solution, evaluating the model's debugging capabilities. The modified solution is re-submitted to the Leetcode IDE for evaluation.
   a) If the modified solution passes all the test cases, the process proceeds to Step 1 for further analysis and assessment.
   b) If the modified solution fails to meet the requirements and triggers any of the error messages encountered in Step 2, the problem is considered a failed attempt by ChatGPT.

TABLE II
DISTRIBUTION OF IMPROVED VS NOT IMPROVED INSTANCES AFTER FEEDBACK

Category	**Percentage**
Improved	36.7%
Not Improved	63.3%

IV. PROMPT ENGINEERING

We used GPT-4 in the zero-shot setting. Rigorous prompt engineering was performed to achieve the best possible output and two prompts for the two parts of the experiment were selected. For generating the initial solution, we outlined the 'Problem Definition', 'Examples', 'Constraints', and 'Code Structures' and directed the model to return the solution according to the code structures. This type of prompting was essential to receive precise responses from the model. For the debugging task, we provide the compiler output along with the erroneous solution while prompting. The final version of the prompts is given in figure 2 and figure 3.

V. ANALYSIS OF CHATGPT'S PERFORMANCE

We proceed to evaluate the performance of ChatGPT in generating the solutions to the programming challenges from natural language problem descriptions. The results obtained provided valuable statistical insights into the model's capabilities.

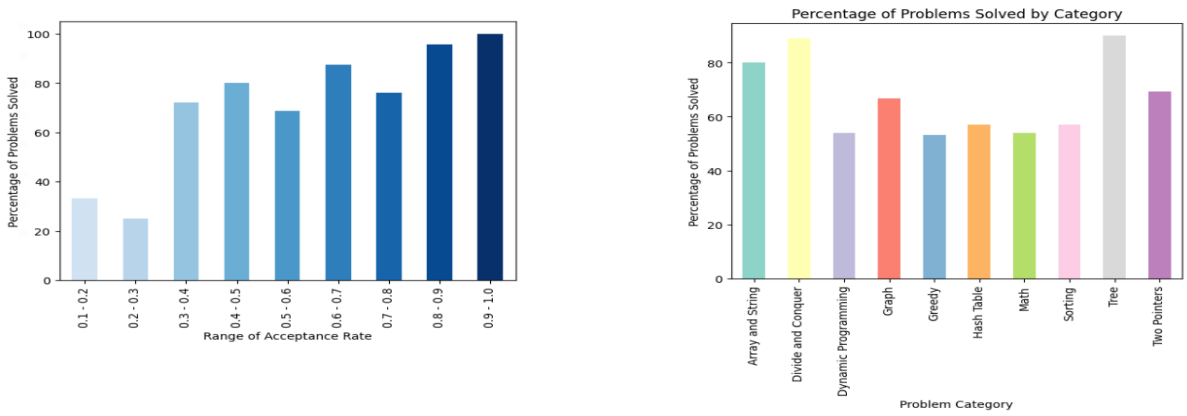

Fig. 4. Analysing the percentage of problems solved concerning the acceptance rate demonstrates a positive correlation between the acceptance rate of problems and the percentage of successful solutions generated by ChatGPT. In terms of the performance of ChatGPT across various problem domains, as indicated by pass rates, reveals that the model excels most in addressing 'Divide and Conquer' and 'Tree' problems, while lower proficiency in tackling 'Greedy' and 'Dynamic Programming (DP)' problems.

Overall Performance

ChatGPT exhibited an overall success rate of **71.875%** across the entire dataset. This indicates that out of 128 problems, ChatGPT successfully generated solutions for 92 of them. Notably, among these successful cases, 84 problems were solved in the initial attempt, rest 8 were solved after prompting the ChatGPT to debug the previously provided solution with the feedback received from Leetcode. There were 36 problems within the dataset for which ChatGPT did not produce satisfactory solutions, even after revisiting the problems with feedback from Leetcode.

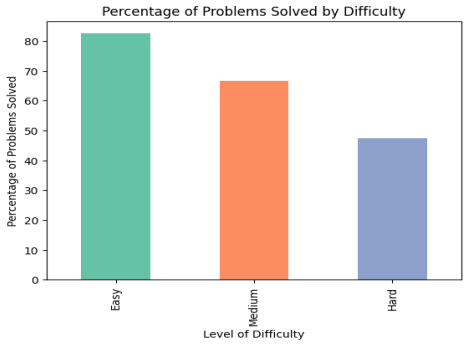

Fig. 5. For problems with lower difficulty, a high rate of success in solving the problems is noticed. However, the success rate declines as the problem starts being more difficult, dropping the success rate by almost 35%.

Feedback and Debugging

Out of the 36 problems for which ChatGPT initially failed to produce correct solutions, Leetcode provided error messages and feedback, along with a few failed test cases, to prompt ChatGPT to rectify its errors. However, even with this additional feedback, ChatGPT failed to produce correct solutions in the majority of cases.

When ChatGPT attempted to fix its errors using the provided feedback, the new solutions exhibited a downgrade in performance. Table II shows that in approximately 63% of the instances, these revised solutions failed to pass previously passed test cases, indicating a decrease in solution correctness. Only around 36% of the time did the new solutions perform better, passing more test cases than before, but still not achieving complete accuracy.

ChatGPT's inability to produce correct solutions even with error messages highlights its limitations in debugging. This reveals a weakness in the model's capacity to learn from and correct errors based on external input, limiting its effectiveness in enhancing solution accuracy, as evidenced by the diminished performance of revised solutions.

Across Different Domains

Analyzing ChatGPT's success rate across different problem domains reveals intriguing findings, as shown in Figure 4. The model achieves the highest success rates on problems from the 'Tree' and 'Divide and Conquer' domains, while exhibiting subpar performance on problems from the 'Greedy' and 'Dynamic Programming (DP)' domains.

This observation suggests that ChatGPT performs better on problems with well-defined rules and structured patterns, as they are easier for the model to understand and generate solutions for. Conversely, the model finds it more challenging to solve problems that require deeper analysis and do not conform to specific solution-generating rules, leading to lower success rates.

Across Different Difficulty levels and Acceptance Rates

A similar problem trend can be observed from Figure 5, which provides insights into ChatGPT's success rate in solving problems of varying acceptance rates. Notably, the model

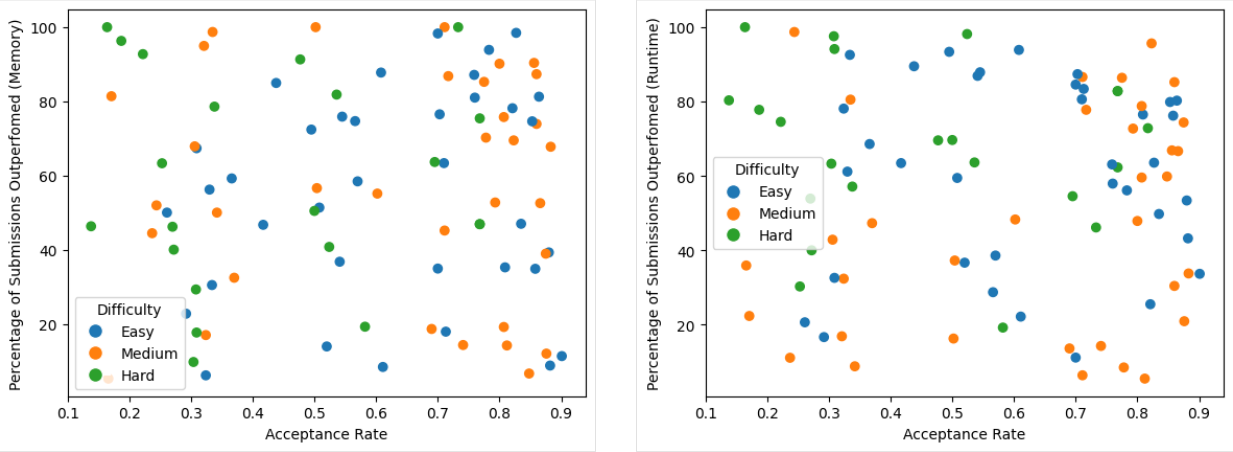

Fig. 6. The two scatter plots reveal that easy problems correlate with higher acceptance rates and efficiency in runtime and memory for ChatGPT, while medium difficulty shows no clear trend, and surprisingly, hard problems with lower acceptance rates also exhibit higher efficiency.

encountered challenges when tackling problems labeled as "Hard", achieving a success rate of 55% in generating accurate solutions. Conversely, when confronted with problems labeled as "Easy", ChatGPT demonstrated a notably higher success rate of 90%. This observation reveals a downward trend in the percentage of successfully solved problems as the difficulty level increases. This phenomenon can be attributed to the fact that "Easy" problems often involve well-established techniques and require less analytical ability, while more challenging problems necessitate deeper analysis and novel problem-solving approaches, which may exceed ChatGPT's current capabilities.

Additionally, ChatGPT's performance is influenced by the acceptance rate of the problems. The acceptance rate reflects the success rate of all submitted solutions for a particular problem on the Leetcode platform. It is noteworthy that the dataset used in this study does not encompass problems with acceptance rates below 10% or above 90%. When analyzing problems with lower acceptance rates, particularly below 30%, ChatGPT struggled to produce accurate solutions, resulting in success rates ranging from 30% to 40%. Conversely, as the acceptance rate increased, ChatGPT exhibited higher success rates, achieving an impressive 95.65% success rate when the acceptance rate exceeded 80%.

Runtime and Memory Efficiency

Let us shift our focus from the broader dataset to a more specific subset comprising instances where ChatGPT successfully solved problems.

Figure 6 illustrates ChatGPT's comparative performance in solving problems, focusing on time and memory efficiency across difficulty levels. The scatterplot reveals that "Easy" problems are predominantly situated towards the right, indicating a higher user success rate compared to "Hard" problems, which are more frequently found towards the left. This aligns with the intuitive expectation that easier problems are more accessible to a wider audience.

Furthermore, "Easy" problems are more likely to be clustered in the upper half of the scatterplot, suggesting that ChatGPT is more adept at crafting efficient solutions for problems of lesser complexity. In contrast, "Hard" problems often require more sophisticated strategies like dynamic programming, making it difficult for ChatGPT to generate these approaches. As a result, ChatGPT does not perform as well in outperforming most submissions in terms of time and memory efficiency for "Hard" problems. For "Medium" difficulty problems, ChatGPT performs fairly well, with no discernible pattern in the diagrams.

VI. Conclusion

ChatGPT represents a significant advancement in AI-driven code generation, demonstrating a high success rate across diverse programming problems. Its proficiency in tackling structured problem domains and the correlation between its success rate and problem acceptance rates highlight its capabilities. However, ChatGPT does not consistently produce the most efficient solutions in terms of runtime or memory usage, and it generates inaccurate solutions approximately 30% of the time, which it fails to debug even with feedback from Leetcode. This underscores the model's limitations in certain problem scenarios. Despite these limitations, this exploration into ChatGPT's capabilities emphasizes its potential to revolutionize code generation and assist programmers. It also highlights the need for continuous refinement and evolution to enhance the capabilities of LLMs like ChatGPT, paving the way for more efficient, intuitive, and powerful coding assistance in the future.

References

[1] H. Tian, W. Lu, T. O. Li, X. Tang, S. C. Cheung, J. Klein, & T. F. Bissyandé (2023). Is ChatGPT the Ultimate Programming Assistant–How far is it?. arXiv preprint arXiv:2304.11938.

[2] R. Bavishi, C. Lemieux, R. Fox, K. Sen, & I. Stoica (2019). AutoPandas: neural-backed generators for program synthesis. Proceedings of the ACM on Programming Languages, 3(OOPSLA), 1-27.

[3] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, ... & O. Vinyals (2022). Competition-level code generation with alphacode. Science, 378(6624), 1092-1097.

[4] M. L. Siddiq, S. H. Majumder, M. R. Mim, S. Jajodia, & J. C. Santos (2022, October). An Empirical Study of Code Smells in Transformer-based Code Generation Techniques. In 2022 IEEE 22nd International Working Conference on Source Code Analysis and Manipulation (SCAM) (pp. 71-82). IEEE.

[5] A. Svyatkovskiy, S. K. Deng, S. Fu, & N. Sundaresan (2020, November). Intellicode compose: Code generation using transformer. In Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (pp. 1433-1443).

[6] X. Hu, G. Li, X. Xia, D. Lo, & Z. Jin (2018, May). Deep code comment generation. In Proceedings of the 26th conference on program comprehension (pp. 200-210).

[7] B. Li, M. Yan, X. Xia, X. Hu, G. Li, & D. Lo (2020, November). DeepCommenter: a deep code comment generation tool with hybrid lexical and syntactical information. In Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (pp. 1571-1575).

[8] S. Stapleton, Y. Gambhir, A. LeClair, Z. Eberhart, W. Weimer, K. Leach, & Y. Huang (2020, July). A human study of comprehension and code summarization. In Proceedings of the 28th International Conference on Program Comprehension (pp. 2-13).

[9] W. Wang, Y. Zhang, Y. Sui, Y. Wan, Z. Zhao, J. Wu, ... & G. Xu (2020). Reinforcement-learning-guided source code summarization using hierarchical attention. IEEE Transactions on Software Engineering, 48(1), 102-119.

[10] R. Gupta, S. Pal, A. Kanade, & S. Shevade (2017, February). Deepfix: Fixing common c language errors by deep learning. In Proceedings of the AAAI Conference on Artificial Intelligence (Vol. 31, No. 1).

[11] N. Jiang, T. Lutellier, Y. Lou, L. Tan, D. Goldwasser, & X. Zhang (2023). Knod: Domain knowledge distilled tree decoder for automated program repair. arXiv preprint arXiv:2302.01857.

[12] Y. Li, S. Wang, & T. N. Nguyen (2022, May). Dear: A novel deep learning-based approach for automated program repair. In Proceedings of the 44th International Conference on Software Engineering (pp. 511-523).

[13] H. Ye, M. Martinez, & M. Monperrus (2022, May). Neural program repair with execution-based backpropagation. In Proceedings of the 44th International Conference on Software Engineering (pp. 1506-1518).

[14] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, ... & I. Polosukhin (2017). Attention is all you need. Advances in neural information processing systems, 30.

[15] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. D. O. Pinto, J. Kaplan, ... & W. Zaremba (2021). Evaluating large language models trained on code. arXiv preprint arXiv:2107.03374.

[16] J. Devlin, M. W. Chang, K. Lee, & K. Toutanova (2018). Bert: Pre-training of deep bidirectional transformers for language understanding. arXiv preprint arXiv:1810.04805.

[17] H. Tian, K. Liu, A. K. Kaboré, A. Koyuncu, L. Li, J. Klein, & T. F. Bissyandé (2020, December). Evaluating representation learning of code changes for predicting patch correctness in program repair. In Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering (pp. 981-992).

[18] T. Zhang, B. Xu, F. Thung, S. A. Haryono, D. Lo, & L. Jiang (2020, September). Sentiment analysis for software engineering: How far can pre-trained transformer models go?. In 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME) (pp. 70-80). IEEE.

[19] S. Lertbanjongngam, B. Chinthanet, T. Ishio, R. G. Kula, P. Leelaprute, B. Manaskasemsak, ... & K. Matsumoto (2022, October). An Empirical Evaluation of Competitive Programming AI: A Case Study of AlphaCode. In 2022 IEEE 16th International Workshop on Software Clones (IWSC) (pp. 10-15). IEEE.

[20] Y. Gu, R. Tinn, H. Cheng, M. Lucas, N. Usuyama, X. Liu, T. Naumann, J. Gao, & H. Poon (2021). Domain-specific language model pretraining for biomedical natural language processing. ACM Transactions on Computing for Healthcare (HEALTH), 3(1), 1-23.

[21] E. Kasneci, K. Seßler, S. Küchemann, M. Bannert, D. Dementieva, F. Fischer, ... & G. Kasneci (2023). ChatGPT for good? On opportunities and challenges of large language models for education. Learning and Individual Differences, 103, 102274.

[22] OpenAI, Available: https://openai.com/

[23] GPT-4, Available: https://openai.com/gpt-4

[24] Leetcode, Available: https://leetcode.com/

[25] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. D. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, & others (2021). Evaluating large language models trained on code. arXiv preprint arXiv:2107.03374.

[26] Z. Chen, W. Chen, C. Smiley, S. Shah, I. Borova, D. Langdon, ... & W. Y. Wang (2021). Finqa: A dataset of numerical reasoning over financial data. arXiv preprint arXiv:2109.00122.

[27] M. M. A. Syeed, M. Farzana, I. Namir, I. Ishrar, M. H. Nushra, & T. Rahman (2022, June). Flood Prediction Using Machine Learning Models. In 2022 International Congress on Human-Computer Interaction, Optimization and Robotic Applications (HORA) (pp. 1-6). IEEE.

[28] A. B. Abacha, & P. Zweigenbaum (2015). MEANS: A medical question-answering system combining NLP techniques and semantic Web technologies. Information processing & management, 51(5), 570-594.

[29] R. Dabre, C. Chu, & A. Kunchukuttan (2020). A survey of multilingual neural machine translation. ACM Computing Surveys (CSUR), 53(5), 1-38.

[30] L. Abualigah, M. Q. Bashabsheh, H. Alabool, & M. Shehab (2020). Text summarization: a brief review. Recent Advances in NLP: the case of Arabic language, 1-15.

[31] B. Chen, F. Zhang, A. Nguyen, D. Zan, Z. Lin, J. G. Lou, & W. Chen (2022). Codet: Code generation with generated tests. arXiv preprint arXiv:2207.10397.

[32] A. Radford, K. Narasimhan, T. Salimans, & I. Sutskever (2018). Improving language understanding by generative pre-training.

[33] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, & I. Sutskever (2019). Language models are unsupervised multitask learners. OpenAI blog, 1(8), 9.

[34] A. M. Dakhel, V. Majdinasab, A. Nikanjam, F. Khomh, M. C. Desmarais, & Z. M. J. Jiang (2023). Github copilot ai pair programmer: Asset or liability?. Journal of Systems and Software, 203, 111734.

[35] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, & others (2020). Language models are few-shot learners. arXiv preprint arXiv:2005.14165.

[36] J. A. Prenner, & R. Robbes (2021). Automatic Program Repair with OpenAI's Codex: Evaluating QuixBugs. arXiv preprint arXiv:2111.03922.

[37] OpenAI (2022). ChatGPT: Optimizing language models for dialogue. https://openai.com

[38] D. Sobania, M. Briesch, C. Hanna, & J. Petke (2023). An analysis of the automatic bug fixing performance of chatgpt. arXiv preprint arXiv:2301.08653.

[39] C. S. Xia, & L. Zhang (2023). Conversational automated program repair. arXiv preprint arXiv:2301.13246.

[40] B. Chen, F. Zhang, A. Nguyen, D. Zan, Z. Lin, J. G. Lou, & W. Chen (2022). Codet: Code generation with generated tests. arXiv preprint arXiv:2207.10397.