2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE) | 979-8-3503-2996-4/23/$31.00 ©2023 IEEE | DOI: 10.1109/ASE56229.2023.00096

A Closer Look at Different Difficulty Levels Code Generation Abilities of ChatGPT

Dapeng Yan
Nanjing University of Aeronautics and Astronautics
dapeng.yan@nuaa.edu.cn

Zhipeng Gao*
Zhejiang University
zhipeng.gao@zju.edu.cn

Zhiming Liu
Southwest University
zliu@nwpu.edu.cn

Abstract*—Code generation aims to generate source code implementing human requirements illustrated with natural language specifications. With the rapid development of intelligent software engineering, automated code generation has become a hot research topic in both artificial intelligence and software engineering, and researchers have made significant achievements on code generation. More recently, large language models (LLMs) have demonstrated outstanding performance on code generation tasks, such as ChatGPT released by OpenAI presents the fantastic potential on automated code generation. However, the existing studies are limited to exploring LLMs' ability for generating code snippets to solve simple programming problems, the task of competition-level code generation has never been investigated. The specifications of the programming competition are always complicated and require the specific input/output format as well as the high-level algorithmic reasoning ability. In this study, we conduct the first large empirical study to investigate the zero-shot learning ability of ChatGPT for solving competition programming problems. Specifically, we warm up the design of prompts by using the Human-Eval dataset. Then, we apply the well-designed prompt to the competition-level code generation dataset, namely APPS, to further explore the effectiveness of using ChatGPT for solving competition problems. We collect ChatGPT's outputs on 5,000 code competition problems, the evaluation results show that it can successfully pass 25.4% test cases. By further feeding extra information (e.g, test failed information) to ChatGPT, we observe that ChatGPT has the potential to fix partial pass into a fully pass program. Moreover, we investigate the solutions generated by LLMs and the existing solutions, we find that it prefers to directly copy the code instead of re-write when facing more difficult problems. Finally, we evaluate the code quality generated by ChatGPT in terms of "code cleanness", we observe that the generated codes are with small functions and file sizes, which are in line with the standard of clean code.

***Index Terms*—code generation, program competition, ChatGPT, large language model, clean code**

I. Introduction

In the context of continuous and agile iteration in software development, there is a growing realization that historical code can be reused to facilitate high-quality and efficient software development [1]. Reusing clean and easy-to-read code can streamline the entire development process and significantly reduce the costs associated with later maintenance [2].

One prominent technology for code reusability is automatic code generation, where code is automatically generated with developers' requirements by accessing vast code bases. The recent research on intelligent code generation has witnessed a surge, indicating its increasing significance in academia and industry. Over the years, researchers have explored automatic code generation using various technologies. For instance, Ling et al. [3] employed natural language descriptions of specific cards to automatically generate corresponding code definitions, thus reducing developers' time and effort writing card effects. Building on the top of Ling et al.'s work, Yin et al. [4] improved the approach by incorporating grammar modeling of the target language as prior knowledge during training. Contemporary, Iyer et al. [5] compressed all code idioms into a simplified syntax tree with a depth of 2 to produce more accurate code.

In 2020, Feng et al. introduced CodeBERT [6], a large-scale bimodal pre-training model for multiple programming languages, which gained the highest popularity at that occasion in the field of intelligent code generation. CodeBERT is widely used to encode input text or code, which is applied to various downstream tasks, including code generation and retrieval. Subsequently, Guo et al. [7] proposed GraphCodeBERT, which incorporates word masking in text information and random data node masking in the data flow graph of code to improve downstream task performance efficiently. Towards the end of 2021, OpenAI released CodeX [8], a large-scale model pre-trained on public datasets based on GPT-3. The Copilot [1] plug-in based on CodeX has become a benchmark for code generation auxiliary tools. The HumanEval dataset proposed in the CodeX paper also serves as a widely used benchmark dataset for subsequent code generation tasks. Recently, Tian et al. [9] conducted an empirical study to evaluate ChatGPT's capability as a programming assistant using two LeetCode programming datasets. Contemporary, Nascimento et al. [9] conducted an empirical investigation to compare the performance of AI systems and software engineers.

Previous works in code generation have primarily focused on simple code generation tasks but neglected the potential of large language models (LLMs), such as ChatGPT, in solving complex coding problems, especially competition programming problems. Moreover, their approaches often directly invoke ChatGPT's APIs to solve problems, with limited optimization methods and relatively simplistic or subjective problem classifications in the dataset. Compared to solving

*Corresponding author.

[1] https://github.com/features/copilot/

2643-1572/23/$31.00 ©2023 IEEE
DOI 10.1109/ASE56229.2023.00096

Authorized licensed use limited to: Zhejiang University. Downloaded on July 02,2024 at 11:24:12 UTC from IEEE Xplore. Restrictions apply.

Problem statement of Maximal Binary Matrix: You are given matrix with n rows and n columns filled with zeroes. You should put k ones in it in such a way that the resulting matrix is symmetrical with respect to the main diagonal (the diagonal that goes from the top left to the bottom right corner) and is lexicographically maximal. One matrix is lexicographically greater than the other if the first different number in the first different row from the top in the first matrix is greater than the corresponding number in the second one. If there exists no such matrix then output -1.	
Input example	Output example
2 1	1 0 0 0

Fig. 1: Example of program competition problem statement.

straightforward coding problems like "*find the maximum element from a list*", competition-level coding problems are significantly more intricate, demanding a comprehensive understanding of algorithms, data structures, and optimization techniques, etc., such as the example of competition programming, the ``Maximal Binary Matrix" Problem, depicted in Figure 1. These challenges often involve multiple steps and intricate problem-solving skills.

Given the vast capabilities of LLMs, such as ChatGPT, it is reasonable to explore their potential in effectively solving those challenging coding problems, potentially even outperforming experienced and skillful human coders. This paper presents the first large-scale empirical study to investigate ChatGPT's ability[2] to generate code solutions for coding problems varying difficulty levels. Additionally, we explore the potential of using feedback information to improve the performance and quality of code generation. Specifically, we collect ChatGPT outputs for 5,000 competition-level coding problems and evaluate ChatGPT's effectiveness in code generation from three key aspects: the accuracy of generated code solutions, the similarity between generated code solutions and the ground-truth ones, the overall code quality of the generated solutions.

The main contributions of this work include:

- A preliminary study that reveals the significant impact of a well-designed prompt on improving the accuracy of the solution code generated by ChatGPT. With our re-designed prompt, the accuracy of ChatGPT is increased from 48.1% to 65.6% when applying the GPT-3.5 model.
- We conduct the first large-scale empirical study assessing ChatGPT's ability to solve competition-level coding problems. Using our well-designed prompt, we run ChatGPT on the APPS dataset, comprising 5,000 competition problems with varying difficulty levels. The experimental results indicate that ChatGPT achieves a 13.1% strict accuracy in solving competition-level problems, while its accuracy for introductory problems is increased to 30.1%.
- We explore the potential of teaching ChatGPT to revise partial pass solutions into fully passable ones by providing additional information, such as failed test cases. Through several rounds of interactions with ChatGPT, solutions generated by ChatGPT can be further optimized, even for the most challenging competition problem.
- The similarity analysis between the accurate results provided by ChatGPT and the ground truths reveals a higher proportion of code reuse in the generated code, particularly for more straightforward problems.
- We select measurable metrics from the "Clean Code" principles to analyze the accurate solution codes generated by ChatGPT, and the findings indicate overall high code quality. Additionally, we provide a replication package [10] of this study to facilitate other researchers.

[2]https://openai.com/blog/chatgpt/

The organization of this paper is as follows. Section II describes the background of code generation. In Section III, we present the key motivations and research questions that drive our study. The details of our research approach are outlined in Section IV. Section V presents the findings and results of our study. In Section VII, we discuss the potential threats to the validity of our work, while Section VIII provides an overview of the key related works. Finally, in Section IX, we draw conclusions and summarize the main contributions of this paper.

II. Background

Code generation has become a subject of significant academic interest due to its potential to enhance software development automation. Numerous scholars discuss different code generation methods, drawing from their professional expertise and practical experience in automating the production of highly accurate code.

The main concepts and fundamental components of code generation methods can be categorized into three groups: code generation methods combined with retrieval, approaches linked with post-processing, and methods relying on code features [11]. The first two categories are advancements over the initial code generation methods. In contrast, the last category can be subdivided into code generation methods based on supervised learning and code generation methods based on pre-training, depending on the employed paradigms.

In the code generation method **based on code features**, the first approach is centered on **supervised learning**, and a commonly used model is the sequence-to-sequence model [12]. This model follows an encoder-decoder paradigm comprising two main parts: the encoder and the decoder. For instance, in 2019, Sun et al. [13] introduced a grammar-based structured Convolutional Neural Network (CNN) that completes code generation tasks by adhering to grammatical construction rules within the abstract syntax tree. Similarly, in the same year, Wei et al. [14] adopted a dual-task learning approach to

Authorized licensed use limited to: Zhejiang University. Downloaded on July 02,2024 at 11:24:12 UTC from IEEE Xplore. Restrictions apply.

enhance code generation and code summarization performance simultaneously.

The second category involves the use of pre-training models. This approach entails pre-training the model on a large-scale unlabeled dataset and fine-tuning it on a downstream labeled dataset using the pre-trained representation. Nijkamp et al. [15] released the CodeGEN pre-training model, boasting a substantial 16.1B model parameters. It underwent training on three datasets: THEPILE[3], BIGQUERY [4], and BIGPYTHON. The use of this pre-training model led to significant performance improvements in code generation compared to single-round methods, effectively validating the effectiveness of the conversational code generation paradigm. InCoder [16], on the other hand, departed from the traditional left-to-right code generation pre-training model paradigm. It introduced a generative model that could perform program synthesis (by generating from left to right) and editing (by masking and filling), leveraging bidirectional context to enhance code generation task performance significantly.

The code generation method **combined with retrieval** involves aiding the decoder in generating code by retrieving similar codes, thereby reducing the decoding space and ultimately enhancing the quality of the generated code [17]–[20]. Hayati et al. [17] were the first to introduce retrieval technology into the code generation task, proposing the "RECODE" model. This model used the sequence of generated behaviors to alter the probability of specific behaviors of syntax tree construction in the final decoding process, resulting in improved model performance. Xu et al. [18] addressed the lack of natural language annotations and code data by utilizing the best-performing "TRANX" model [19]. They incorporated two external knowledge bases for data enhancement, aiding the model in pre-training and improving its performance.

The **combined with post-processing** approach relies on large-scale pre-trained language models. The critical focus is testing the model generation process and the generated results using test samples to improve performance [21]–[23]. Some works directly utilize test examples to enhance the model's performance. Jain et al. [21] introduced a post-processing module to conventional large-scale pre-trained language models. This module checks the syntax and semantics of the code, ensuring that the generated code successfully passes test samples and other quality checks. Despite ongoing advancements in code generation models, most deep learning-based methods still struggle to guarantee the compilability of generated code. To address this issue, Wang et al. [22] proposed "COMPCODER," a method that enhances the model's ability to generate compilable code. They employ compilation signals novelty and design a new approach to train generators and discriminators for compilable code generation tasks simultaneously.

In contrast to the works mentioned above, our empirical study investigates the code generation ability of a large model (ChatGPT) using a more complex natural language dataset. Additionally, we integrate an automatic repair process into the code generation flowchart, optimizing the accuracy of solution codes to ensure they pass all test cases. Our analysis also includes an assessment of the quality of these generated codes.

[3]https://pile.eleuther.ai/

[4]https://cloud.google.com/bigquery

TABLE I: Experimental results with HumanEval.

Dataset	Count	No Response	Fail	Pass	Pass ratio
Human-eval	164	5	51	108	65.6%

III. Preliminary Study

Before conducting the sizeable empirical study on competition-level code generation tasks, we conducted a preliminary study on the widely-studied dataset HumanEval [8] that comprises 164 Python programming questions, and each of them consists of a function header, a function body, and several test cases (averaging 7.7 test cases per question). These questions assess language understanding, algorithms, and simple mathematics, resembling basic programming interview questions. To make ChatGPT understand the given questions better and generate code solutions in batches, we optimize the prompt design and invoke ChatGPT's API to automatically generate code solutions by asking it with an additional sentence at the front of each question: "Please generate the Python solution code for the following question." Subsequently, we use the test suits in HumanEval to validate whether the generated code solutions can successfully pass all tests. The experimental results are presented in Table I.

In this experiment, we set the temperature of ChatGPT as 0 to ensure that the generated code solutions are consistent. The results show that ChatGPT achieves an accuracy rate of 65.6%, significantly surpassing the 48.1% accuracy reported by OpenAI [24] for the same model. This observation highlights the substantial impact of employing a suitable prompt in enhancing the accuracy of generating code solutions with ChatGPT.

Inspired by the potential of ChatGPT in solving simple programming problems, we embark on an exploration to determine how developers can effectively collaborate with LLMs. To achieve this, it is essential to understand the code generation capabilities and limitations of LLMs (e.g., correctness and code quality) for simple code and complex coding tasks. Competition-level coding problems encompass various categories such as number theory, graph theory, algorithmic game theory, data structures, computational geometry, and string analysis, demanding a significantly higher level of cognitive reasoning and problem-solving proficiency.

IV. Methodology

This section presents the research questions, object selection, and measurement selection to explore the capability of ChatGPT in generating code for solving competition-level problems.

Authorized licensed use limited to: Zhejiang University. Downloaded on July 02,2024 at 11:24:12 UTC from IEEE Xplore. Restrictions apply.

A. Research Questions

Our investigation addresses the following research questions (RQs):

- **RQ-1.** ***To what extent can ChatGPT automatically generate accurate code for solving complex competition problems?*** ChatGPT can automatically generate program code based on prompts describing programming tasks. We focus on assessing the accuracy of ChatGPT in generating the correct code for solving competition-level problems.
- **RQ-2.** ***Are solutions generated by ChatGPT comparable to ground truth at different difficulty levels?*** We investigate the generation habits of ChatGPT by examining the similarity between strict accurate codes generated by ChatGPT and the ground truth for problems with varying difficulty levels.
- **RQ-3.** ***How clean is the code generated by ChatGPT?*** We examine whether ChatGPT can automatically generate clean code efficiently, addressing developers' need for high-quality code generation.

B. Object Selection

Model Selection. Language models pre-trained on large-scale corpora have demonstrated impressive capabilities in solving various natural language processing tasks. Researchers have explored the scaling effect by increasing the model's scale and found that larger language models possess significantly greater capacity and outperform small-scale models, such as BERT [25], [26]. The continuous advancements in large language models have been boosted by academia and industry. As an AI chatbot built upon large language models, ChatGPT has garnered widespread attention and created a significant milestone in the progress of intelligent technology.

In this study, we opt to use ChatGPT as our experimental model. Specifically, we utilize the "`text-davinci-003`" variant, which falls under the GPT-3.5 category. With an impressive parameter count of over 175 billion, this model is one of the largest and most powerful language models available today, making it highly suitable for various coding-related tasks. Regarding the parameter settings of using this model, the "`temperature`" parameter determines the sampling temperature, ranging from 0 to 2, with higher values leading to more random outputs. In our experiment, we set it as 0 to ensure stable and controllable output results. Additionally, we consider the "`top_p`" parameter, also known as kernel sampling, which considers the probability mass of labels. For our experiment, we set it as 1.

Dataset Selection. For this study, we select the APPS dataset [27], which consists of 10,000 coding problems sourced from various program competition platforms, including Codewars [5], AtCoder [6], Kattis [7], and Codeforces [8], and offers a diverse range of problems, spanning from simple to highly challenging competition tasks. Each problem within the APPS dataset is accompanied by a set of ground-truth solutions and, on average, 21.1 test cases. As illustrated in Table II, the APPS dataset stands out for its substantial number of programming competition problems, test cases for validating the correctness of generated code solutions, and available ground-truth solutions for reference. Compared with other code-generation datasets like PY150 [28] and CONCODE [5], the APPS dataset specifically focuses on competition-level code-generation tasks, making it an ideal choice for our investigation.

TABLE II: Details of commonly used datasets.

	APPS	SPoC	PY150	CONCODE
Test Cases	**Yes**	Yes	None	None
# of Problems	**10,000**	677	3,000	104,000
Language	**Python**	C++	Python	Java

APPS contains two kinds of data, i.e., training and test data. The test data specifies each problem with the corresponding test cases, which satisfies the requirement in our experiments of using test cases to validate the correctness of the generated code. So we select the 5,000 problems in the test data as the dataset of this work. Furthermore, the ChatGPT model we selected has a token limitation of 4,096. When the tokens in both the prompt input and the answer exceed this threshold, calling ChatGPT's API to generate code will be ceased and fail. A total of 515 questions in APPS test data cannot be successfully processed by ChatGPT due to this token limitation. We remove these 515 questions from our dataset. Eventually, 4,485 questions are selected as the final dataset for this study, of which details are presented in Table III.

TABLE III: Details of the selected dataset.

Difficulty	Dataset		Actual use	
	Question	Test Case	Question	Test Case
Introductory	1,000	11,747	966	11,040
Interview	3,000	78,382	2,668	70,671
Competition	1,000	15,830	851	13,872
Total	5,000	105,959	4,485	95,583

C. Measurement Selection

For RQ-1, we employ the accuracy rate and pass ratio metrics. The accuracy rate represents the proportion of solutions generated by ChatGPT that can successfully pass all test cases. The pass ratio, also known as pass@1 [8], measures the percentage of test cases passed by the generated solutions. It is a widely adopted evaluation metric for assessing the performance of auto-generated programs.

To answer RQ-2, we calculate the textual similarity between the strict accurate solution code provided by ChatGPT and the ground truth solutions. To perform this calculation, we utilize the similarity function from the `text2vec` package in Python. As some problems have multiple ground truth solutions, we analyze the distribution of similarities for each problem and explore different difficulty levels based on two other methods: the highest similarity and the average similarity.

[5]https://www.codewars.com/
[6]https://atcoder.jp/
[7]https://open.kattis.com/
[8]https://codeforces.com/

Authorized licensed use limited to: Zhejiang University. Downloaded on July 02,2024 at 11:24:12 UTC from IEEE Xplore. Restrictions apply.

For RQ-3, we investigate the cleanness of the accurate codes generated by ChatGPT. Code quality is evaluated using the metrics mentioned in Martin's book [29]. However, not all indicators from the book can be automatically assessed. In this study, we select two commonly used **measurable indicators** for analysis: function size and file size.

V. Study Results

A. RQ-1: Effectiveness of ChatGPT Solutions

To answer the research question RQ-1, we investigate the effectiveness of the code generated by GPT-3.5 for solving competition-level problems. A crucial aspect of using GPT-3.5 effectively is the design of appropriate prompts that align with the tasks; the appropriate prompt can significantly enhance the generated code's performance and quality. The selection of an appropriate prompt depends on various factors, including the specific task, input data, and desired output format. A well-crafted prompt should offer clear and concise instructions to the model, guiding it toward producing the expected output while minimizing the risk of generating irrelevant or misleading results. Our experiment consists of two phases to enhance the accuracy of the generated code.

First Phase. The initial step involves using GPT-3.5 to generate a code solution for a single iteration, referred to as \`"Round-1", which we preserve for subsequent assessment and enhancement. The APPS dataset problem encompasses two input formats: call-based format and standard input format. As a result, we design distinct prompts for each input format. For the **standard input format**, we design the following prompt:

* *Act as a python developer. You will be given a problem, which contains an overview of the problem, a description of the input and output, and some examples of input and output. Your job is to write well-formed Python code to solve this problem, which can pass all inputs and expect outputs given in the problem example. In addition, the format you respond to is very strict, and please use the call-based format to write the code without comments. The detail of the question is the following:* `[QUESTION_PLACEHOLDER]`.

For the **call-based format** problems, we add an extra sentence to the end of the prompt of standard input format:

* *The function should be defined as [starter_code_str].*

During this phase of prompt design, our objective is twofold: ❶ to acquaint GPT-3.5 with its intended task of solving programming problems, and ❷ to facilitate its understanding of the problem's structure (e.g., through examples of input/output) and the expected output (e.g., Python code to handle all inputs and produce desired outputs). This approach aims to enhance GPT-3.5's ability to generate effective solutions.

Second Phase. With the first phase, GPT-3.5 can generate preliminary code solutions for all the competition coding problems. However, many code solutions either cannot pass the corresponding test cases or partially pass them. For a given question with a set of test cases T, the corresponding solution generated by GPT-3.5 can pass a set of test cases T', where $T' \subset T$, we group such solution as **partially-passed** one. If T' is empty, the generated code solution is a failed solution. Thus, in the second phase, we aim to explore the feasibility of teaching GPT-3.5 to revise partially passed/failed solutions to accepted (correct) solutions. To this end, we employ a standardized prompt that provides additional information to GPT-3.5, enabling it to address the areas of failure in these solutions:

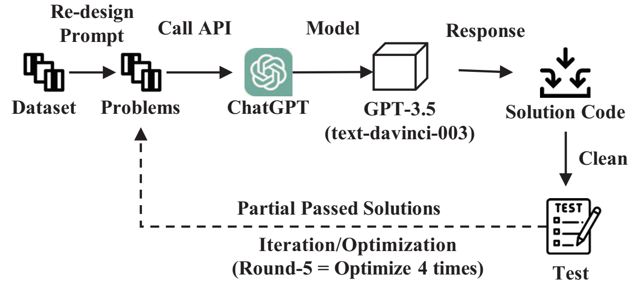

Fig. 2: Workflow of generating solutions automatically.

* *I asked you to act as a Python developer to help us solve the following question before: [QUESTION]. The solution you gave us before is: [LAST_SOLUTION]. However, the code can not pass all testing inputs and outputs, which are: [INPUT_OUTPUT]. Please modify your solution code to ensure it can pass all testing inputs and outputs.*

In this phase, our attention is directed solely toward the generated partially-passed solutions. We posit that if a solution fails to pass any test case, it signifies that the competition problem is too intricate for GPT-3.5 to solve. Given this circumstance, repeating the generation process would result in significantly low efficiency and prohibitively high costs. Consequently, for each partially-passed solution, we perform four additional iterations and select the solution that achieves the highest number of successful test cases among the five generated solutions (including the first partially-passed solution), designating it as the final solution (`Round-5`). The overall workflow of our approach is illustrated in Figure 2.

Data Cleaning. In the experiment, we observe that even though we request the GPT-3.5 API to generate pure Python code without any redundant parts, the generated code solutions still contain a mixture of natural language notes (e.g., markdowns and others). Here, we outline the common answer formats we encountered and subsequently remove them to prevent testing errors:

* *In the form of Markdown*
* *Begin with "Solution:"*
* *Begin with "—–Python Code—–"*
* *Begin with "——Solution—–"*

The performance of GPT-3.5 is presented in Table IV, wherein we compare its preliminary results (`Round-1` results) with GPT-Neo and a more recent model (CodeRL) [30] on APPS from two aspects: the average and strict accuracy of test cases across various difficulty levels. It should be noted that GPT-Neo and CodeRL did not redesign the prompt during the

Authorized licensed use limited to: Zhejiang University. Downloaded on July 02,2024 at 11:24:12 UTC from IEEE Xplore. Restrictions apply.

TABLE IV: The comparison of GPT-3.5, GPT-Neo and CodeRL on the average percentage of test cases passed and strict accuracy for different difficulty levels.

	Test Case Average				Strict Accuracy			
Model	Introductory	Interview	Competitive	Average	Introductory	Interview	Competitive	Average
GPT-Neo	14.7%	9.9%	6.5%	10.2%	3.9%	0.6%	0.0%	1.1%
CodeRL	-	-	-	-	7.1%	1.9%	0.8%	2.7%
GPT-3.5	37.6%	25.3%	16.1%	25.4%	30.1%	9.7%	4.3%	13.1%

verification test. Notably, GPT-3.5 significantly outperforms GPT-Neo, achieving a test case average and strict accuracy of 37.6% and 30.1%, respectively, for the introductory level, in contrast to GPT-Neo's 14.7% and 3.9%. Regarding competition-level problems, GPT-Neo fails to generate any correct code solutions, while CodeRL-generated code solutions can only pass 0.8% of the problems. In contrast, GPT-3.5 successfully solves 4.3% of the most challenging programming problems (competition-level) , and its generated code solutions can pass 16.1% of the test cases. For the other difficulty levels, GPT-3.5 also achieves the highest pass rate. It is essential to note that GPT-Neo is meticulously trained and fine-tuned on the APPS dataset, whereas GPT-3.5 successfully tackles these competition problems through its zero-shot learning ability without any specific training process. This underscores the potential of GPT-3.5 for competition-level code generation tasks. Moreover, it demonstrates the superior performance of GPT-3.5 in understanding and solving simple and complex problems.

TABLE V: Comparison of Top-5 and Round-5 results on APPS by evaluating the performance of GPT-Neo and GPT-3.5 on introductory problems.

Top/Round-5 Results	GPT-Neo	GPT-3.5
Test Case Average	19.9%	25.7%
Strict Accuracy	5.5%	14.7%

Furthermore, we compare the results of GPT-3.5 `Round-5` with GPT-Neo Top-5 on introductory level problems, illustrated in Table V. We omitted CodeRL for comparison here because they did not give the detailed pass ratio of test cases in their research. The Top-5 best performance of GPT-Neo in terms of "Test Case Average" and "Strict Accuracy" are 19.9% and 5.5%, respectively, compared to 25.7% and 14.7% for GPT-3.5, both of which significantly surpass the state-of-the-art [27] experimental results. It is worth mentioning that for GPT-3.5 `Round-5`, we consider partially passed problems, representing the lower bound of GPT-3.5's results. Furthermore, we find that the percentage improved of strict accuracy(14.7%-5.5%=9.2%) is higher than the test case average(25.7%-19.9%=5.8%), and those problems with more test cases are hard to be repaired after five rounds to become the strict accuracy solution. Therefore, problems with fewer test cases are easier to fix under partial pass.

We have observed variations in the test pass rates and accuracy across problems of different difficulty levels. Specifically, the introductory-level problems exhibit a higher test pass rate, which is reasonable as these problems are generally easier to understand and solve than the other difficulty levels. To assess

TABLE VI: Average passing rate for solutions that pass partial test cases.

	Introductory	Interview	Competition
Round-1	39.9%	32.5%	34.7%
Round-5	38.3%	32.1%	35.0%

the robustness of GPT-3.5 in solving competition problems of different levels, we further calculated the average test pass rate for partially-passed solutions across different difficulty levels, as presented in Table VI. The table shows that the overall average test pass rate is approximately 35%, indicating that GPT-3.5 provides relatively consistent responses when it comprehends the problem, irrespective of the difficulty level. However, the average test pass rates for the introductory and interview levels slightly decreased from `Round-1` to `Round-5`, suggesting that in the subsequent iterations, GPT-3.5 either fully understands the problem and provides accurate responses, or it still generates imprecise answers that do not significantly increase the number of test cases passed.

```
--- a/chatgpt_solution.txt
+++ b/opt_gptsolution_2.py
@@ -1,10 +1,12 @@
+# Modified Solution
 n = int(input())
 a = list(map(int, input().split()))
 mod = 10**9 + 7
 ans = 0
-m = 1
-for i in range(2, n+1):
-    if a[m-1] < a[i-1]:
-        ans += a[m-1]
-        m = i
+for perm in itertools.permutations(a):  # Iterate
    through all permutations of the array a
+    m = 1  # Set m to 1 for each permutation
+    for i in range(2, n+1):  # Iterate through each
     element of the permutation
+        if perm[m-1] < perm[i-1]:  # Check if the
    current element is greater than the previous one
+            ans += perm[m-1]  # If so, add the
    previous element to the answer and set m to i
+            m = i
 print(ans % mod)
```

Fig. 3: The patch diff for problem "1506" in the test folder of `APPS`.

To gain a deep insight into the generated code by GPT-3.5, we selected two examples that initially passed only partial tests in the first round but successfully passed all tests in the subsequent optimization process for analysis. The first example is the question "1506" with a difficulty level of interview, accessible through the link here. According to the problem statement, "two elements are considered different if their indices differ, so for every array, there are exactly n!

Authorized licensed use limited to: Zhejiang University. Downloaded on July 02,2024 at 11:24:12 UTC from IEEE Xplore. Restrictions apply.

permutations." This problem requires calculating all permutations of a sequence. To compare the code generated in the first round with the optimized code, we present their diff information in Figure 3. The preliminary solution only calculates the sequence once, leading to only 4 of the 16 test cases passing. However, after feeding the failed tests back to GPT-3.5, the optimized solution successfully calculates the entire permutation of the input sequence, resulting in all 16 test cases passing. In other words, GPT-3.5 did not fully capture the entire permutation requirement in the first round. However, it comprehended the requirement during the second round after we "taught" GPT-3.5, where test cases failed.

```
--- a/chatgpt_solution.txt
+++ b/opt_gptsolution_2.py
@@ -3,7 +3,16 @@ sticks = list(map(int, input().
    split()))
 if N < 3:
     print("impossible")
 elif N == 3:
-    if (sticks[0] + sticks[1] > sticks[2]) and (
    sticks[1] + sticks[2] > sticks[0]) and (sticks
    [2] + sticks[0] > sticks[1]):
+    if (sticks[0] + sticks[1] > sticks[2]) and (
    sticks[1] + sticks[2] > sticks[0]) and (sticks
    [2] + sticks[0] > sticks[1]) and (sticks[0] !=
    0) and (sticks[1] != 0) and (sticks[2] != 0):
         print("possible")
     else:
-        print("impossible")
\ No newline at end of file
+        print("impossible")
+else: # N > 3
+    for i in range(N-2): # Iterate through all
    possible combinations of three sticks.
+        for j in range(i+1, N-1):
+            for k in range(j+1, N):
+                if (sticks[i] + sticks[j] > sticks[
    k]) and (sticks[j] + sticks[k] > sticks[i]) and
    (sticks[k] + sticks[i] > sticks[j]) and (sticks
    [i] != 0) and (sticks [j] != 0) and (sticks [k]
    != 0):   # Check if the three selected lengths
    can form a triangle.
+                    print("possible")   # If so,
    output "possible".
+                    exit()   # Exit the program.
+
+    print("impossible") # If no combination of
    three lengths can form a triangle, output "
    impossible".
```

Fig. 4: The patch diff for problem "4895" in the test folder of APPS.

Another example is the problem "4895" with a difficulty level of introductory, which can be accessed through the link here. This problem involves solving the "Sticky Situation" programming problem, which entails determining whether a triangle can be formed using several sticks of different lengths. In other words, the task is to arrange and combine each group of sticks and judge whether they meet the conditions for establishing a triangle. Similarly, we present the diff information before and after GPT-3.5 revises the code in Figure 4. The initial code only handles cases where the number of sticks is less than 3 or equal to 3, but it does not consider cases where the number of sticks is greater than 3. After GPT-3.5 optimizes the code, it adds processing for cases where the number of sticks is greater than 3 and introduces three layers of for loops to obtain all possible permutations and combinations, enabling the judgment of triangle establishment conditions. Additionally, the optimized code includes an early exit from the loop if a triangle can be formed, thereby avoiding redundant calculations. As a result, the optimized code exhibits more comprehensive processing and successfully passes all test cases.

✍ Answers to RQ-1

We conduct the first comprehensive empirical study to evaluate the potential of GPT-3.5 in solving programming problems of different difficulty levels. Our experimental results demonstrate that GPT-3.5 significantly outperforms the best model on this task, showcasing its remarkable zero-shot learning ability.

B. RQ-2: Similarities between ChatGPT solutions and ground-truth solutions

Building upon our observations from the first research question, we aim to investigate whether ChatGPT demonstrates varying behaviors (copying from somewhere else or writing by itself) when solving problems of different difficulty levels. To achieve this, we analyze the similarity between the code solutions generated by ChatGPT and the ground truth provided in the dataset, mainly focusing on the solutions that passed all test cases in the `Round-5` results.

For the problems in the APPS dataset, each problem may have multiple ground truth solutions. To analyze the similarity between the ground truth solutions and their corresponding ChatGPT-generated solutions, we pair them together and calculate the similarity score. The distribution of these similarity scores is illustrated in Figure 5. The mean and median values of these similarities are 78.2% and 81.0%, respectively, with the majority of results falling within the range of [69.7%, 89.2%]. This indicates that the solution codes generated by ChatGPT exhibit a high degree of overlap with the codes in the ground truth solutions.

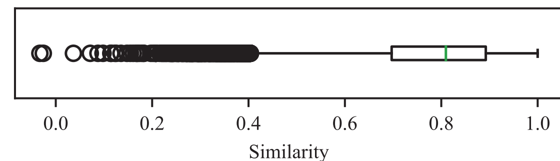

Fig. 5: The distribution of similarities between accurate ChatGPT solutions and their corresponding ground truths.

We then investigate the similarities for different difficulty levels and employ two methods to obtain the final similarity scores. The first approach calculates the average of all similarities for each problem, while the second approach selects the maximum similarity score. During this investigation, we obtained 659 ChatGPT solutions that ultimately passed the test cases from the experimental results in Section V-A. However, we encountered difficulties for 146 of these solutions as the corresponding ground truth was not available, leading to a total

Authorized licensed use limited to: Zhejiang University. Downloaded on July 02,2024 at 11:24:12 UTC from IEEE Xplore. Restrictions apply.

TABLE VII: The distribution of similarities.

Similarity(%)	Average	Maximum
[0-20]	0	0
(20-40]	0	0
(40-60]	39	11
(60-80]	280	56
(80-100]	194	446

of 513 actual computed final similarity scores. Subsequently, we classified these scores into different intervals to study their distribution, as presented in Table VII.

Table VII reveals that the average and maximum similarity scores are not lower than 40%, with most scores concentrated above 60%. Only a few similarities fall between 40% and 60%. However, when considering the maximum similarity score, an impressive 87% of the similarities are concentrated above 80%.

TABLE VIII: Average similarities for each difficulty level.

Difficulty	Introductory	Interview	Competition
Average	79.5%	72.1%	65.3%

We calculated the average similarity scores for each difficulty level of the problems to assess the performance of ChatGPT, and the results are presented in Table VIII. It is evident that as the problems become more challenging, the average similarity decreases, while for simpler problems, the average similarity increases. This finding indicates that as the complexity of the problem increases, ChatGPT tends to compose the code by itself rather than copying directly from the code bases. Furthermore, we examined the maximum similarity for each question and found that 9 interviews and 22 introductory problems achieved a 100% similarity, thereby confirming our earlier observations.

```
=====ChatGPT Solution======
n = int(input())
a = list(map(int, input().split()))
lcm_min = float('inf')
i_min, j_min = 0, 0
for i in range(n):
    for j in range(i+1, n):
        lcm_ij = (a[i]*a[j])
        if lcm_ij < lcm_min:
            lcm_min = lcm_ij
            i_min, j_min = i+1, j+1
print(i_min, j_min)
```

```
======== Solution 1 =========
import math from collections
import defaultdict
import sys
input = sys.stdin.readline
def main():
    n = int(input())
    a = list(map(int, input().split()))
    MAX = 10**7 + 1
    res = MAX * MAX
...
    res = MAX * MAX
    for index in range(n):
        val = a[index]
...
def __starting_point():
    main()
__starting_point()
```

Fig. 6: Comparison between the ChatGPT solution and ground truth of problem "4066".

We conducted a case study to gain deeper insights into the reasons behind these high similarities, selecting two examples with the highest and lowest similarities below 100%. Firstly, we focused on problem No. "4066" that has only one ground truth solution, resulting in a similarity of 53.9% between it and the ChatGPT solution. This similarity arises due to the similar definitions in front of the code; however, their methods are different. Figure 6 displays the codes of these two solutions. In the ChatGPT solution on the left, the variable names such as i_min and j_min hold meaningful values used to record the minimum index. Moreover, this solution relies on fewer third-party libraries, resulting in cleaner and more generalized code. Notably, it uses **"inf" to initialize the minimum value, proving more efficient than the** "MAX" calculation. As a result, the ChatGPT-generated code contains significantly fewer tokens, indicating reduced potential bugs and enhanced readability. Furthermore, this ChatGPT solution successfully passes all 32 test cases for the problem, affirming its higher quality in terms of correctness and functionality.

We proceeded to investigate another case randomly chosen. The problem is "1701" and has 24 ground truth solutions. The highest similarity of 92.7% was calculated with solution 14. Figure 7 presents both the ChatGPT-generated solution and solution 14 details. Compared to the traditional implementation, a clear advantage of the ChatGPT code is its enhanced readability and more detailed comments. Moreover, naming local variables in the ChatGPT solution is more meaningful, making it easier to understand their purpose. For instance, the variable "servers" in the ChatGPT code indicates that it stores content related to all IP addresses and corresponding servers. In contrast, the variable "d" in the ground truth code is less intuitive. Additionally, using "command" on the left is more readable than "s" on the right. Furthermore, the ChatGPT code is refined, employing unpacking syntax for "command" and "ip" assignments, allowing the loop to read the input and automatically assign values to corresponding variables simultaneously. This approach is more efficient than the four lines of code on the right.

```
=====ChatGPT Solution ======
n, m = map(int, input().split())
servers = {}
for _ in range(n):
    name, ip = input().split()
    servers[ip] = name
for _ in range(m):
    command, ip =
input().split(';')[0].split()
    print(f'{command} {ip};
#{servers[ip]}')
```

```
======== Solution 14 ========
n,m=map(int,input().split())
d={}
for _ in range(n):
  name,ip=input().split()
  d[ip]=name
for _ in range(m):
  s=input()
  ip=s.split()[-1]
  if ip[-1]==';':
    ip=ip[:-1]
  print(s,"#"+d[ip])
```

Fig. 7: Comparison between the ChatGPT solution and ground truth of problem "1701".

✍ Answers to RQ-2

We observed that on simpler problems, ChatGPT tends to reuse more existing code, while on more complicated problems, it prefers to create new solutions from scratch. Additionally, the code produced by ChatGPT adheres more closely to programming rules and is easier to read.

Authorized licensed use limited to: Zhejiang University. Downloaded on July 02,2024 at 11:24:12 UTC from IEEE Xplore. Restrictions apply.

C. RQ-3: Cleanness of ChatGPT solutions

In this study, we investigate the strict accurate solutions generated by ChatGPT from four aspects: function size, file size, and length of code lines, to provide measurable indicators for assessing the quality of ChatGPT-generated code.

Function sizes are assessed based on the cyclomatic complexity[9] and lines of code (LOCs) in each function. Additionally, **file sizes** are evaluated based on the lines of code in each source file. Furthermore, the number of code tokens in each line determines the **code line length**. Uncle Bob emphasized the importance of small functions with meaningful names for better clean code practice [29]. According to Google's recommendations presented in Table IX, developers should limit each C, C++, JavaScript, and Python code line to 80 characters and each Java code line to 100 characters [32]. McCabe's presentation on "Software Quality Metrics to Identify Risk" [33] interprets cyclomatic complexity in four categories. Functions with a cyclomatic complexity ranging from 1 to 10 have simple procedures with little risk. On the other hand, functions with higher cyclomatic complexity will result in more complicated code with higher risk.

TABLE IX: Recommended LOCs and cyclomatic complexity of functions.

Google	C	C++	Java	JavaScript	Python
LOCs of functions	40	-	-	-	40
Cyclomatic Complexity					
1 - 10	Simple procedure, little risk				
11 - 20	More complex, moderate risk				
21 - 50	Complex, high risk				
> 50	Untestable code, very high risk				

1) Function Sizes: Figure 8 illustrates the distribution of function sizes regarding cyclomatic complexity. Since many solutions generated by ChatGPT do not have explicitly defined functions, only 34 functions are included in the statistics. In comparison, 68,448 functions from the ground truth dataset are participating in the statistics. Most Python code functions generated by ChatGPT satisfy the criterion of small functions in clean code, with their cyclomatic complexities limited to within 5.

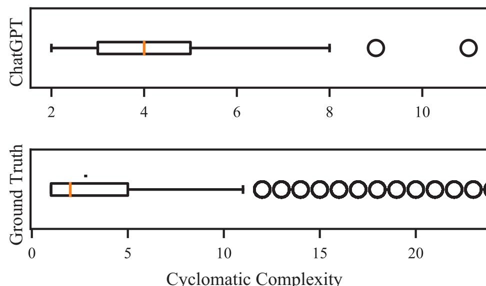

Fig. 8: Distribution of cyclomatic complexities.

[9]Cyclomatic complexity is a measurable indicator representing the number of linearly independent paths in a section of the source code. It was developed by Thomas J. McCabe Sr [31] to describe the complexity of a program.

Figure 9 displays the distribution of Lines of Code (LOCs) for functions. Most of these functions contain 19 lines or fewer, significantly below Google's standard of 40 lines. The central box portions of the distribution are [7, 14] and [3, 16], respectively, indicating that the lower quartile of ChatGPT-generated code surpasses that of the ground truth. In contrast, the upper quartile falls below it. This demonstrates that the distribution of ChatGPT-generated code is more concentrated, with its overall distribution being lower than that of the ground truth.

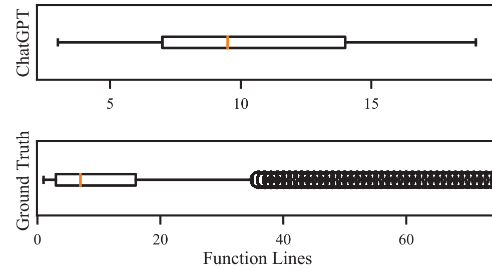

Fig. 9: Distribution of function lines.

2) File Sizes: Figure 10 illustrates the distribution of file sizes in terms of Lines of Code (LOCs) for the source code files. We considered 659 accurate ChatGPT-generated solution files and 88,014 ground-truth solutions from the APPS dataset for analysis. Both solutions predominantly contain code within 26 lines (the third quartile), suggesting their suitability for solving competition-level problems rather than more extensive project requirements.

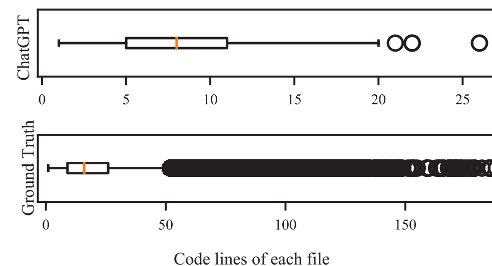

Fig. 10: Distribution of file lines.

Comparing the box sizes, we observe that ground truth solutions encompass a much wider range of file sizes than ChatGPT solutions. Overall, file size distribution aligns with the pattern seen in function sizes. ChatGPT-generated solutions are distributed between 5 and 11 lines, whereas ground truth solutions span [9, 26] lines. Consequently, the distribution of ChatGPT-generated solutions is smaller than the ground truth.

ChatGPT's solution code employs smaller file sizes to address various problems, including introductory, interview, and competition problems within the APPS dataset.

Authorized licensed use limited to: Zhejiang University. Downloaded on July 02,2024 at 11:24:12 UTC from IEEE Xplore. Restrictions apply.

✍ **Answers to RQ-3**

The distributions of function size and file size of the accurate solutions generated by ChatGPT, measured by the code quality based on the standard of clean code, are comparable to the ground truth solutions. This indicates that ChatGPT can generate code as clean as professional developers produce.

VI. Implication

Implications for researchers: Our paper verifies the feasibility of using LLMs to solve competition-level programming problems by iteratively calling the ChatGPT API to repair partially passed code. However, the success rate is still suboptimal, and the performance can be further boosted by involving developers' feedback and/or code contextual information (e.g., the abstract syntax tree, the control flow graph, or the data flow graph). Furthermore, generating higher-quality code through ChatGPT, aligned with clean code principles, poses potential and widespread research directions for software engineering researchers.

Implications for practitioners: One of the essential takeaways from our study is the great potential of using ChatGPT for solving coding problems. For the developers who need to write algorithms to solve their daily problems, our study suggests that by describing the details of the problem, the ChatGPT can provide possible solutions to their problems, which can significantly improve problem-solving efficiency and save developers' effort on more creative works. Practitioners could consider adopting LLMs in their daily development for solving coding problems.

VII. Threats to Validity

Threats to external validity: One threat to external validity in our code generation analysis is the model selection, as GPT-4 is expensive, and our application to access GPT-4 API is still pending. To mitigate this threat, we opt for the earlier version of the model, ChatGPT-3.5. While its performance may be less advanced than GPT-4, we ensure consistency by comparing the results obtained using the same model in our study. Another potential threat to external validity lies in the training data of ChatGPT, which might already include the problems under investigation. To address this concern, we conduct a preliminary study using HumanEval to explore this potential bias. Moreover, due to the high token cost of re-throwing the question to ChatGPT each time, we choose only to regenerate codes for incomplete pass solutions. Instead, we focus on optimizing codes that can partially pass the test cases to proceed with our experimental step.

Threats to internal validity: Our study's internal validity threats arise from adjusting model parameters to generate solutions. Randomly generated results can be challenging to optimize effectively. To address this concern, we mitigate the threat by fixing the parameters, ensuring consistent effects for each problem. Another validity concern relates to analyzing similarities between ChatGPT and ground truth. Since text similarity alone may not fully reflect code consistency, we perform case analysis to reduce subjective judgment errors. In the future, we aim to address this issue by incorporating code cloning technology. Additionally, our evaluation of code cleanliness focuses on measurable indicators. However, clean code encompasses various aspects, such as design rules, code understandability tips, function implementation rules, and writing comment guidelines, which warrant qualitative analysis. These metrics will be considered in our future work.

VIII. Related Work

Code Generation Evaluation. Researchers employ a variety of indicators to quickly evaluate the quality of generated code using a unified standard. One of the most commonly used metrics is exact match accuracy, which represents the percentage of exact matches between the model-generated code and the reference code. While BLEU [34] is a widely used index for evaluating text/code generation quality [35]–[38], its application in code generation technology is limited. Ren et al. [39] argue that the BLEU index, designed for natural language evaluation, overlooks the grammatical and semantic aspects of code, making it unsuitable for code evaluation. To address this limitation, a new evaluation index called CodeBLEU is introduced. CodeBLEU incorporates n-gram matching advantages from BLEU and integrates code syntax through the abstract syntax tree (AST) and code semantics through the data stream.

LLMs' code generation capability. In 2017, Chen et al. [40] introduced Codex, a GPT language model fine-tuned on publicly available code from GitHub, explicitly focusing on Python code-writing capabilities. The performance of Codex on typical introductory programming problems was analyzed by J Finnie-Ansley et al. [41], who compared its results with those of students who took the same exams under normal conditions. Notably, Codex outperformed most students. Additionally, in 2023, Liu et al. [42] proposed EvalPlus, an evaluation framework that precisely assesses the functional correctness of LLM-generated code. In the same year, Arghavan Moradi Dakhel et al. [43] research the capabilities of Copilot by generating solutions for fundamental algorithmic problems, and they compare Copilot's proposed solutions with those of human programmers on a set of programming tasks. They find that Copilot can provide solutions to most problems. However, some solutions are buggy and non-reproducible. However, these studies needed more practical verification with diverse, complex real-world problems. In contrast, our work systematically evaluates ChatGPT's capabilities using a wide range of practical and challenging tasks, providing a more comprehensive and practical assessment.

IX. Conclusion

The ability of large models for automatic code generation has been extensively explored in the research community, with various approaches to analyze the correctness of generated solutions. In this work, we introduce a novel method involving re-designed prompts to enhance the accuracy of codes generated by ChatGPT. Notably, this aspect needs to

Authorized licensed use limited to: Zhejiang University. Downloaded on July 02,2024 at 11:24:12 UTC from IEEE Xplore. Restrictions apply.

be thoroughly explored in the existing literature. To validate the effectiveness of our approach, we conduct a preliminary study using a relatively simple dataset (Human-Eval). The positive results from this analysis demonstrate the feasibility of our method. Subsequently, we extend our investigations to a more complex dataset (APPS) to comprehensively assess the generation ability of large models like ChatGPT across problems of varying difficulty levels. Furthermore, we delve into the generating habits of ChatGPT and evaluate the quality of the generated code. Our experimental findings highlight that a well-designed prompt significantly enhances the accuracy of code generation. Specifically, for simple problems, the solutions generated by ChatGPT exhibit more remarkable similarity to the ground truth. Moreover, the strict and accurate solutions provided by ChatGPT also demonstrate comparable code quality from the perspective of clean coding practices, similar to code written by professional developers. These observations underscore the potential of large models like ChatGPT in aiding code generation tasks, mainly when guided by optimized prompts.

ACKNOWLEDGMENT

This research is partially supported by the Shanghai Rising-Star Program (23YF1446900) and the National Science Foundation of China (No. 62202341). This research is partially supported by the Starry Night Science Fund of Zhejiang University Shanghai Institute for Advanced Study, Grant No. SN-ZJU-SIAS-001.

REFERENCES

[1] Z. Gao, X. Xia, D. Lo, and J. Grundy, "Technical q&a site answer recommendation via question boosting," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 30, no. 1, pp. 1–34, 2020.

[2] Z. Yang, S. Chen, C. Gao, Z. Li, G. Li, and R. Lv, "Deep learning based code generation methods: A literature review," *CoRR*, vol. abs/2303.01056, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2303.01056

[3] W. Ling, E. Grefenstette, K. M. Hermann, T. Kočiskỳ, A. Senior, F. Wang, and P. Blunsom, "Latent predictor networks for code generation," *arXiv preprint arXiv:1603.06744*, 2016.

[4] P. Yin and G. Neubig, "A syntactic neural model for general-purpose code generation," *arXiv preprint arXiv:1704.01696*, 2017.

[5] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, "Mapping language to code in programmatic context," *arXiv preprint arXiv:1808.09588*, 2018.

[6] 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.

[7] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu *et al.*, "Graphcodebert: Pre-training code representations with data flow," *arXiv preprint arXiv:2009.08366*, 2020.

[8] 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.

[9] H. Tian, W. Lu, T. O. Li, X. Tang, S. Cheung, J. Klein, and T. F. Bissyandé, "Is chatgpt the ultimate programming assistant - how far is it?" *CoRR*, vol. abs/2304.11938, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2304.11938

[10] "Source code of experiments," https://zenodo.org/record/7899853#.ZFTpkXZBxD8, Last Access: May 2023.

[11] Z. Yang, S. Chen, C. Gao, Z. Li, G. Li, and R. Lv, "Deep learning based code generation methods: A literature review," *arXiv preprint arXiv:2303.01056*, 2023.

[12] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," *Advances in neural information processing systems*, vol. 27, 2014.

[13] Z. Sun, Q. Zhu, L. Mou, Y. Xiong, G. Li, and L. Zhang, "A grammar-based structural CNN decoder for code generation," in *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019*. AAAI Press, 2019, pp. 7055–7062. [Online]. Available: https://doi.org/10.1609/aaai.v33i01.33017055

[14] B. Wei, G. Li, X. Xia, Z. Fu, and Z. Jin, "Code generation as a dual task of code summarization," *Advances in neural information processing systems*, vol. 32, 2019.

[15] L. Phan, H. Tran, D. Le, H. Nguyen, J. Anibal, A. Peltekian, and Y. Ye, "Cotext: Multi-task learning with code-text transformer," *arXiv preprint arXiv:2105.08645*, 2021.

[16] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong, "Codegen: An open large language model for code with multi-turn program synthesis," *arXiv preprint arXiv:2203.13474*, 2022.

[17] S. A. Hayati, R. Olivier, P. Avvaru, P. Yin, A. Tomasic, and G. Neubig, "Retrieval-based neural code generation," *arXiv preprint arXiv:1808.10025*, 2018.

[18] F. F. Xu, Z. Jiang, P. Yin, B. Vasilescu, and G. Neubig, "Incorporating external knowledge through pre-training for natural language to code generation," *arXiv preprint arXiv:2004.09015*, 2020.

[19] P. Yin and G. Neubig, "Tranx: A transition-based neural abstract syntax parser for semantic parsing and code generation," *arXiv preprint arXiv:1810.02720*, 2018.

[20] Z. Gao, X. Xia, D. Lo, J. Grundy, and Y.-F. Li, "Code2que: A tool for improving question titles from mined code snippets in stack overflow," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 1525–1529.

[21] N. Jain, S. Vaidyanath, A. Iyer, N. Natarajan, S. Parthasarathy, S. Rajamani, and R. Sharma, "Jigsaw: Large language models meet program synthesis," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 1219–1231.

[22] X. Wang, Y. Wang, Y. Wan, F. Mi, Y. Li, P. Zhou, J. Liu, H. Wu, X. Jiang, and Q. Liu, "Compilable neural code generation with compiler feedback," *arXiv preprint arXiv:2203.05132*, 2022.

[23] B. Chen, F. Zhang, A. Nguyen, D. Zan, Z. Lin, J.-G. Lou, and W. Chen, "Codet: Code generation with generated tests," *arXiv preprint arXiv:2207.10397*, 2022.

[24] OpenAI, "Gpt-4 technical report," 2023.

[25] J. Devlin, M. Chang, K. Lee, and K. Toutanova, "BERT: pre-training of deep bidirectional transformers for language understanding," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, J. Burstein, C. Doran, and T. Solorio, Eds. Association for Computational Linguistics, 2019, pp. 4171–4186. [Online]. Available: https://doi.org/10.18653/v1/n19-1423

[26] W. X. Zhao, K. Zhou, J. Li, T. Tang, X. Wang, Y. Hou, Y. Min, B. Zhang, J. Zhang, Z. Dong, Y. Du, C. Yang, Y. Chen, Z. Chen, J. Jiang, R. Ren, Y. Li, X. Tang, Z. Liu, P. Liu, J. Nie, and J. Wen, "A survey of large language models," *CoRR*, vol. abs/2303.18223, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2303.18223

[27] D. Hendrycks, S. Basart, S. Kadavath, M. Mazeika, A. Arora, E. Guo, C. Burns, S. Puranik, H. He, D. Song *et al.*, "Measuring coding challenge competence with apps," *arXiv preprint arXiv:2105.09938*, 2021.

[28] "150k python dataset," https://eth-sri.github.io/py150, 2016.

[29] R. C. Martin, *Clean code: a handbook of agile software craftsmanship*. Pearson Education, 2009.

[30] H. Le, Y. Wang, A. D. Gotmare, S. Savarese, and S. C. Hoi, "Coderl: Mastering code generation through pretrained models and deep reinforcement learning," in *NeurIPS*, 2022. [Online]. Available: http://papers.nips.cc/paper_files/paper/2022/hash/8636419dea1aa9fbd25fc4248e702da4-Abstract-Conference.html

[31] T. J. McCabe Sr, "Cyclomatic complexity," *National Bureau of Standards. special Publication. m99*, 1976.

[32] Google, "Google style guides," https://github.com/google/styleguide, Last Access: August 2022.

[33] T. McCabe, "Software quality metrics to identify risk," http://www.mccabe.com/ppt/SoftwareQualityMetricsToIdentifyRisk.ppt, Last Access: August 2022.

Authorized licensed use limited to: Zhejiang University. Downloaded on July 02,2024 at 11:24:12 UTC from IEEE Xplore. Restrictions apply.

[34] K. Papineni, S. Roukos, T. Ward, and W. Zhu, “Bleu: a method for automatic evaluation of machine translation,” in *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics, July 6-12, 2002, Philadelphia, PA, USA*. ACL, 2002, pp. 311–318. [Online]. Available: https://aclanthology.org/P02-1040/

[35] X. Hu, Z. Gao, X. Xia, D. Lo, and X. Yang, “Automating user notice generation for smart contract functions,” in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 5–17.

[36] X. Hu, X. Xia, D. Lo, Z. Wan, Q. Chen, and T. Zimmermann, “Practitioners’ expectations on automated code comment generation,” in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 1693–1705.

[37] Z. Gao, X. Xia, J. Grundy, D. Lo, and Y.-F. Li, “Generating question titles for stack overflow from mined code snippets,” *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 29, no. 4, pp. 1–37, 2020.

[38] Z. Gao, X. Xia, D. Lo, J. Grundy, X. Zhang, and Z. Xing, “I know what you are searching for: Code snippet recommendation from stack overflow posts,” *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 3, pp. 1–42, 2023.

[39] S. Ren, D. Guo, S. Lu, L. Zhou, S. Liu, D. Tang, N. Sundaresan, M. Zhou, A. Blanco, and S. Ma, “Codebleu: a method for automatic evaluation of code synthesis,” *CoRR*, vol. abs/2009.10297, 2020. [Online]. Available: https://arxiv.org/abs/2009.10297

[40] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, “Evaluating large language models trained on code,” *CoRR*, vol. abs/2107.03374, 2021. [Online]. Available: https://arxiv.org/abs/2107.03374

[41] J. Finnie-Ansley, P. Denny, B. A. Becker, A. Luxton-Reilly, and J. Prather, “The robots are coming: Exploring the implications of openai codex on introductory programming,” in *ACE ’22: Australasian Computing Education Conference, Virtual Event, Australia, February 14 - 18, 2022*, J. Sheard and P. Denny, Eds. ACM, 2022, pp. 10–19. [Online]. Available: https://doi.org/10.1145/3511861.3511863

[42] J. Liu, C. S. Xia, Y. Wang, and L. Zhang, “Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation,” *CoRR*, vol. abs/2305.01210, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2305.01210

[43] A. Moradi Dakhel, V. Majdinasab, A. Nikanjam, F. Khomh, M. C. Desmarais, and Z. M. J. Jiang, “Github copilot ai pair programmer: Asset or liability?” *Journal of Systems and Software*, vol. 203, p. 111734, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121223001292

Authorized licensed use limited to: Zhejiang University. Downloaded on July 02,2024 at 11:24:12 UTC from IEEE Xplore. Restrictions apply.