An Evaluation Method for Large Language Models' Code Generation Capability

Haoran Su, Jun Ai, Dan Yu, and Hong Zhang*,
Beihang University, Beijing, China
suhaoran@buaa.edu.cn, aijun@buaa.edu.cn, 18376206@buaa.edu.cn, zh@buaa.edu.cn
*corresponding author

Abstract—Large language models are becoming increasingly popular in various professional fields. One of their applications is providing code suggestions. However, the differences in code generation capabilities of different large language models and the problems they may make in giving code suggestions have not been well studied. This paper proposes a method for evaluating the code generation capabilities of large language models and applies it to several commonly used models, including ChatGPT, Claude, Spark, and Bing AI. Through experimental evaluation and data analysis, we find that search-based large language models, such as Bing AI, exhibit stronger code generation capabilities than pre-trained models, such as ChatGPT, Claude, and Spark. We also find that the current large language models possess strong natural language understanding abilities, and errors in code suggestions are more likely to be due to code problems rather than understanding problems.

Keywords–Evaluation Method, Large Language Model, Code Generation

1. Introduction

Large language model (LLM) is a neural network-based language model that typically consists of billions of weights or more, and is trained on large amounts of unlabeled text using self-supervised or semi-supervised learning techniques [1, 2]. And with the explosion of ChatGPT in recent months, large language models represented by ChatGPT have gradually entered the public eye and started to play an important role in people's lives. Large language models are general-purpose models that, in addition to regular NLP tasks (such as sentiment analysis, named entity recognition, or mathematical reasoning) [1, 3] , and the ability to conduct daily conversations and Q&A with humans, these models have also shown amazing capabilities in some professional fields, such as the ability to perform automatic diagnosis based on medical images [4] and protein structure parsing [5].

This paper focuses on the code capabilities of these large language models. Automatic generation of codes that satisfy user requirements has been a long-standing problem in computer science, namely program synthesis [6]. There have been many studies dedicated to improving the efficiency of programmers through code recommendation, code search, or other types of program synthesis [7–9], among which the main representatives include GitHub Copilot [10] and Microsoft CodeX [11], among others. However, these tools also have certain problems, such as giving incorrect code suggestions [12] and weak code generation capabilities [13]. The powerful natural language understanding and generation capabilities of large language models may improve this situation, but there is no comprehensive evaluation method to measure how well large language models can generate code and the quality of the code suggestions given, which are the main focus of this paper.

The evaluation method proposed in this paper evaluates the code generation capability of large language models in six dimensions, namely code validity, code correctness, code complexity, code reliability, code security, and code readability. Code validity focuses on evaluating whether the code given by the model can run without errors, which is the most basic requirement for the code. Code correctness focuses on evaluating how well the code suggestions given by the model meet the requirements, which is mainly performed through code testing. Code complexity measures whether the code given by the model is too complex, code reliability and security assess the number of bugs and vulnerability issues in the code suggestions and the difficulty in resolving them, and readability measures the difficulty in understanding the code suggestions by the user. To achieve this goal, a dataset for testing the code capabilities of large language model, LLMC Dataset, is proposed. This dataset comprises of 45 coding questions, and these coding questions cover many aspects of code capabilities, including data types and operators, data structures, control structures, string processing and regular expressions, functions and modules, classes and objects, file handling, and exception handling.

To demonstrate the effectiveness and rationality of the proposed evaluation method, we apply it to several commonly used and freely available large language models, including ChatGPT, Claude, Spark, and Bing AI. Our evaluation focused on four questions: Q1) **Are code suggestions from the large language models correct?** Q2) **What about the quality of the code given by the large language models?** Q3) **What categories of code problems do large language models solve better?** Q4) **Are large language models more likely to make understanding problems or code problems?**

Through experimental evaluation and data analysis, we found that search-based large language models, such as Bing AI, exhibit stronger code generation capabilities than pre-training-based models, such as ChatGPT, Claude, and Spark. Fur-

2767-6684/23/$31.00 ©2023 IEEE
DOI 10.1109/DSA59317.2023.00118

thermore, our analysis of error categories revealed that the current large language models possess strong natural language understanding abilities, and errors in code suggestions were more likely to be due to code problems rather than understanding problems.These findings demonstrate the effectiveness and rationality of the evaluation method proposed in this paper.

Section 2 of this paper presents the proposed methodology and the main evaluation metrics. Section 3 describes the LLMC dataset and the main steps of the experiment and evaluation. Section 4 analyzes the collected data and the results of the evaluation, and answers the questions mentioned in the introduction. The last two sections of the paper are devoted to the conclusion and future work.

2. METHODOLOGY

The method proposed in this paper consists of two main steps: obtaining code suggestions from large language models and code evaluation. The framework of the evaluation method is shown in Figure 1.

To obtain code suggestions from large language models, a dataset of code questions (the LLMC dataset in this paper) is constructed and applied to different models. This enables an evaluation of the performance of large language models on various categories of code questions and an analysis of the problems that arise when large language models give code suggestions.

In the code evaluation part, the code output of the models is analyzed based on six different dimensions: Code Validity, Code Correctness, Code Complexity, Code Reliability, Code Security, and Code Readability. Each question is assigned a weight, and the final score is calculated as the sum of the scores for each dimension.

Code Validity. To evaluate the ability of large language models in generating executable code, the generated code is passed into the Python interpreter and executed. If the code can be executed without errors, it is marked as "1", indicating the code's validity, and vice versa as "0". This metric represents the most basic criterion for evaluating the code capability, and measures the ability of the large language model to produce executable code. The maximum score for this metric is 15.

$$Code\ \ Validity\ \ Score = \begin{cases} 15 & , \quad label = 1 \\ 0 & , \quad label = 0 \end{cases} \tag{1}$$

Code Correctness. In order to evaluate whether a large language model provides code suggestions that meet user requirements, each code question is tested against pre-existing test cases provided by LeetCode and designed by the author. The code should be tested using test cases to ensure that it meets the user's requirements and performs as intended. After executing the test cases, the pass rate is calculated to evaluate the code's performance. The pass rate is a critical metric for evaluating the code's performance and is reported as a percentage. If all test cases pass successfully, the code is considered "Correct" and is given a full score out of 30. Conversely, if the test cases run with errors or the output is not as expected, the model is considered to have given the "Wrong Answer" and is given a partial score based on the percentage of test cases that pass.

$$Code\ \ Correctness\ \ Score = \frac{Pass\ \ Rate\,(\%)}{100\%} \times 30 \tag{2}$$

Code Complexity. This evaluation dimension is designed to determine the code's conciseness and complexity. The primary goal is to assess whether the code generated by the model is sufficiently concise and solves the problem without being overly complex. Current code static analysis tools usually set thresholds of 15-25 for cyclomatic complexity and cognitive complexity [17, 18]. For the LLMC dataset, since the lines of code required to solve a problem are usually less than 100, if both the cyclomatic complexity and cognitive complexity of the code suggestion are less than 15, the code is deemed to satisfy the complexity requirement and is awarded full points (out of 10) for this evaluation metric. However, if one of the two complexity metrics exceeds the threshold of 15, all points for this evaluation metric are deducted.

$$Code\ \ Complexity\ \ Score = \begin{cases} 10 & , \quad Cyclomatic\ \ and\ \ Cognitive\ \ Complexity \leq 15 \\ 0 & , \quad Others \end{cases} \tag{3}$$

Code Reliability and ***Code Security***. The reliability of code is commonly assessed based on the number of bugs present in the code and the effort required to address them, whereas the security of code is measured by the number of vulnerabilities present and the effort required to address them. The ability to provide reliable and secure code recommendations is a crucial aspect of the code capabilities of large language models. Therefore, a code recommendation with a reliability rating of A is awarded a full score out of 10. The same scoring guideline is applied to the scoring of code security. Through the evaluation of code reliability and security by the code static analysis tool, the model's code suggestions are ensured to be reliable and safe.

$$Code\ \ Reliability\ \ Score = \begin{cases} 10 & , \quad Rank = A \\ 0 & , \quad Others \end{cases} \tag{4}$$

$$Code\ \ Security\ \ Score = \begin{cases} 10 & , \quad Rank = A \\ 0 & , \quad Others \end{cases} \tag{5}$$

Code Readability. When providing code suggestions, the model is required to add comments to the code and explain the generated code. The comments rate of the code can be measured using a code static analysis tool. To evaluate the accuracy of the model's explanations, a label of "1" is given if the explanation is comprehensive and error-free, "0" if there are errors or if the code is not fully explained. The ability to comment and explain code as proficiently as a human programmer is a crucial aspect of code competence for large language models. Therefore, the maximum score for this evaluation dimension is set at twenty-five. To achieve a full score of twenty-five, the code must have comments rate greater than 10%, and the explanation accuracy label must be 1. If one of these requirements is not met, only a score of 10 can be obtained. If both requirements are not met, no points will be awarded.

$$Code\ \ Readability\ \ Score = \begin{cases} 25 & , \quad Comments\ \ Rate \geq 10\ \ and\ \ Accuracy\ \ label = 1 \\ 10 & , \quad Comments\ \ Rate \geq 10\ \ or\ \ Accuracy\ \ label = 1 \\ 0 & , \quad Others \end{cases} \tag{6}$$

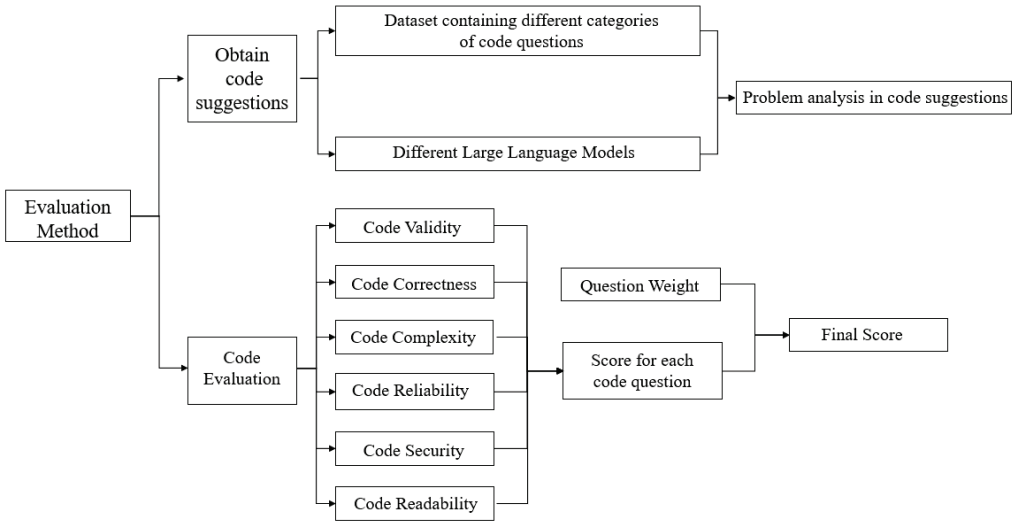

Figure 1. Framework of proposed evaluation method.

Question Weight. To account for the varying difficulty levels of questions in the dataset, each question is assigned a weight, known as the Question Weight. LeetCode categorizes code questions based on their level of difficulty, which can be classified as easy, medium, or hard. It also reports the pass rate of human programmers. Different levels of question weight are established based on the difficulty and pass rate of the questions, as shown in the table below. Questions with higher difficulty and lower pass rates are assigned higher weight because they necessitate more advanced coding skills and are therefore more valuable in evaluating the coding capabilities of large language models. Conversely, easy questions with higher pass rates have lower weight because these problems typically require less complex and simpler code.

Table 1
Question Weight of different questions

Difficulty \ Pass Rate	<40%	40%-70%	>70%
Easy	1.0	0.9	0.8
Medium	1.1	1.0	0.9
Hard	1.2	1.1	1.0

For each problem in the LLMC dataset, the code suggestions generated by the large language model are evaluated in six dimensions, as described above. The score for each question is obtained by adding the scores of the six dimensions, represented as Score*i*. The weight of each question is denoted as Weight*i*. The final score of the model on the LLMC dataset is obtained as follows:

$$Final\ \ Score = \frac{\sum_{i=1}^{45}(Score_i \times Weight_i)}{\sum_{i=1}^{45} Weight_i} \tag{7}$$

The higher the final score, the better the model performed in providing code suggestions. By using a comprehensive evaluation framework and assigning weights to the problems based on their difficulty and pass rates, we are able to accurately and fairly evaluate the code ability of the large language model across a range of programming problems.

3. Experimental Evaluation

The framework of the experimental part of this paper is shown in Figure 2. First, the proposed LLMC dataset is applied to different large language models to obtain their code suggestions. The LLMC dataset contains a diverse set of code questions, covering a range of programming languages and difficulty levels. Then, the experiments are carried out using the evaluation method proposed in Section 2. To facilitate the subsequent evaluation of the models, we also collect data on the performance of each model on each programming question, including the validity score, correctness score, complexity score, reliability score, security score, and readability score, as well as the error causes of incorrect code suggestions.

3.1. LLMC Dataset and SonarQube

LLMC Dataset. In order to evaluate the capability of large language models in coding, the LLMC dataset is created. This dataset consists of 45 code questions collected from LeetCode and designed by the author, covering a wide range of Python fundamentals. LeetCode is a well-known online platform that hosts a repository of over 2,300 code questions with 16 different programming languages, including C, C++, Java, Python and others [14]. The table 2 displays the categories and distribution of questions in the dataset.

Figure 3. shows an example code question from the LLMC dataset. This question defines the concept of a ”good split” and requires the code to compute the number of good splits in the input string ’s’. The question focuses on two aspects: how well the large model understands natural language, i.e., how well it understands the definition of ”good split” in the question, and

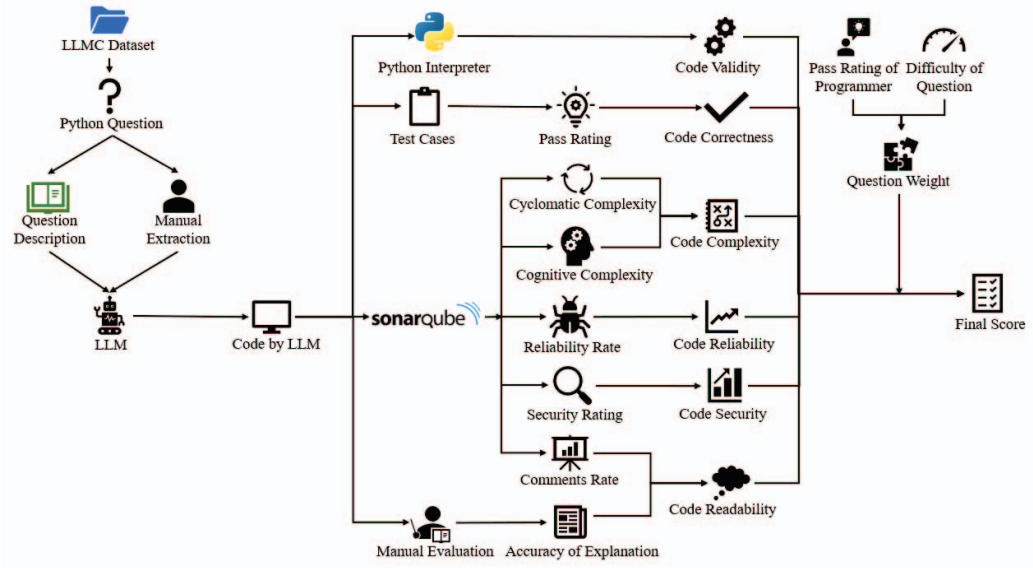

Figure 2. Framework of the experimental part.

Table 2
Questions Categories in LLMC Dataset

Categories	
Data Types and Operators	
Data Structures	Sequence
	Mapping
	Sets
Control Structures	
Strings and Regular Expressions	Strings
	Regular Expressions
Functions and Modules	
Classes and Objects	Classes
	Objects
File Handling	
Exception Handling	

(a) Question Description

(b) Coding Environment of this question

Figure 3. An example code question in the LLMC dataset, named Number of Good Ways to Split a String [12]

how well it can use the code to process strings. Similarly, each question in the dataset focuses on testing some aspect of the performance of the large language model and will be analyzed based on the results once code suggestions are obtained for all questions in the dataset.

SonarQube and ***Code Quality Metrics***. SonarQube is an open-source platform developed by SonarSource for continuous inspection of code quality to perform automatic reviews with static analysis of code to detect bugs and code smells on 29 programming languages. SonarQube offers reports on duplicated code, coding standards, unit tests, code coverage, code complexity, comments, bugs, and security recommendations [15]. The code quality metrics used in this paper include:

- Complexity. Several studies have found that cognitive complexity and cyclomatic complexity have a strong correlation with code comprehensibility and can serve as useful indicators of code complexity[16]. Cyclomatic Complexity measures the number of independent paths through the code. Whenever the control flow of a function splits, the complexity counter gets incremented by one [17]. Cognitive Complexity refers to the mental effort required to understand and reason about the code. The basic rules include [18]:

1) Ignore structures that allow multiple statements to be readably shorthanded into one;
2) Increment (add one) for each break in the linear flow of the code;
3) Increment when flow-breaking structures are nested.

- Reliability and Security. Code reliability refers to the ability of code to remain error-free for a specified period of time, while code security focuses on identifying known vulnera-

bilities in the source code, dependencies, and open source packages [12]. Both reliability and security are critical aspects of software quality and are essential for ensuring the proper functioning and safety of software systems. One way to evaluate code reliability and security is through the use of static code analysis tools like SonarQube. SonarQube can analyze programs without running them to identify any vulnerabilities or bugs in the code. SonarQube provides a variety of reliability and security metrics, including reliability ratings and security ratings, that can be used to assess the reliability and security of code.

Table 3
Reliability Rating and Security Rating criteria of SonarQube

Rating	Reliability	Security
A	0 Bugs	0 Vulnerabilities
B	At least 1 Minor Bug	At least 1 Minor Vulnerability
C	At least 1 Major Bug	At least 1 Major Vulnerability
D	At least 1 Critical Bug	At least 1 Critical Vulnerability
E	At least 1 Blocker Bug	At least 1 Blocker Vulnerability

- Comments Rate. The comments rate of the code can be calculated by the following formula. This metric measures the proportion of commented lines in the code file, and can be used as an indicator of code readability and maintainability [17]. Code with high comment rates is generally easier to understand and modify.

$$Comments\ \ Rate\,(\%) = \frac{Comment\ \ Lines}{(Code\ \ Lines + Comment\ \ Lines)} \times 100\% \tag{8}$$

3.2. Experiment and Evaluation

To evaluate the performance of large language models, we select four examples: OpenAI's ChatGPT[1], Anthropic's Claude[2], iFLYTEC's Spark[3] and Microsoft's Bing AI[4]. ChatGPT ,Claude and Spark are pre-training based generative large language models [20, 21], while Bing AI is a search-based large language model that leverages GPT-4 as its kernel and searches for relevant information from the Internet based on user prompts [19, 21]. By comparing the performance of pre-training based generative large language models and search-based large language models on the LLMC dataset, we can gain preliminary conclusion on which type of model is more capable at providing code suggestions.

The necessary information, including function names, parameters, inputs and outputs, is manually extracted from the questions and passed into the large language model along with the question descriptions to obtain code suggestions from the model. To ensure that the test environment is similar to the user's daily use environment, the descriptions are entered manually instead of calling the API interface of the large language model.

[1]https://chat.openai.com/
[2]https://www.anthropic.com/claude/
[3]https://xinghuo.xfyun.cn/
[4]https://www.bing.com/

After obtaining code suggestions from the large language models, Python scripts are utilized to store them in different files for subsequent evaluation. The code static analysis tool SonarQube 10.0 is employed to report various metrics, including cyclomatic complexity, cognitive complexity, reliability rating, security rating, and comments rate of the generated code. Specifically, the SonarQube API is utilized to save the relevant data.

Finally, we utilize the methodology outlined in Section 2 to calculate the Validity Score, Correctness Score, Complexity Score, Reliability Score, Security Score, and Readability Score of the code suggestions. For Python programming questions, the possible statuses of submitted code are:

- Accepted: The submitted code passes all test cases.
- Wrong Answer: The submitted code differs from the expected output on at least one test case, but it is able to execute successfully without any errors.
- Runtime Errors: The submitted code cannot be executed due to errors on at least one test case.

4. Evaluation Results and Discussion

Based on Equation 7, the Final Score of the four models used for the experimental evaluation is calculated as shown in the following table:

Table 4
Final Scores of different Large Language Models

Large Language Model	Final Score
ChatGPT	84.36
Claude	77.83
Spark	58.61
Bing AI	87.12

As shown in the table, Bing AI achieves the highest score among the three models, with a final score of 87.12. ChatGPT achieves a final score of 84.36, while Claude achieves a final score of 77.83 and Spark of 58.61. Overall, all four models achieves good scores, indicating their potential for generating high-quality code.

Q1) **Are code suggestions from the large language models correct?**

Table 5 displays the test case pass rate for all questions in the dataset for the models. Among the models, Bing AI achieves the highest average pass rate, passing 708 test cases out of a total of 867, resulting in a pass rate of 82%. ChatGPT passes 659 test cases, resulting in a pass rate of 74%. Claude passes 457 test cases with a pass rate of 52 %, and Spark only passes 188 test cases, resulting in a pass rate of 22%.

These results suggest that Bing AI has a higher success rate in generating code that passes the given test cases, compared to ChatGPT, Claude and Spark. However, it is important to note that the pass rate is just one metric for evaluating the performance of large language models, and should be considered in conjunction with other metrics such as code complexity and readability.

Table 5
Correctness of defferent Large Language Models' code suggestions for each question in LLMC Dataset and overall status

Question		Test Cases	Pass Rate			
			ChatGPT	Claude	Spark	Bing AI
Data Types and Operators	1-1	25	100%	100%	0%	100%
	1-2	18	100%	100%	0%	100%
	1-3	20	100%	100%	5%	100%
	1-4	15	67%	100%	0%	100%
	1-5	20	100%	0%	25%	100%
	1-6	24	100%	0%	0%	100%
Data Structures	2.1-1	18	67%	72%	44%	83%
	2.1-2	20	100%	100%	10%	55%
	2.1-3	18	83%	100%	28%	100%
	2.1-4	24	63%	21%	8%	0%
	2.1-5	21	81%	0%	0%	0%
	2.2-1	30	80%	100%	93%	100%
	2.2-2	12	100%	100%	100%	100%
	2.2-3	18	0%	0%	0%	100%
	2.3-1	23	87%	0%	0%	100%
	2.3-2	16	25%	63%	38%	81%
	2.3-3	18	100%	0%	0%	100%
Control Structures	3-1	21	100%	100%	100%	100%
	3-2	20	80%	0%	0%	0%
	3-3	22	100%	100%	0%	100%
String Processing and Regular Expressions	4.1-1	16	100%	0%	0%	100%
	4.1-2	20	0%	0%	0%	0%
	4.1-3	18	100%	0%	0%	100%
	4.1-4	20	80%	0%	0%	65%
	4.2-1	20	100%	100%	0%	100%
	4.2-2	16	25%	25%	0%	81%
	4.2-3	24	100%	100%	42%	100%
Functions and Modules	5-1	16	0%	0%	0%	0%
	5-2	22	64%	59%	55%	86%
	5-3	20	100%	55%	0%	100%
	5-4	24	54%	17%	8%	75%
Classes and Objects	6.1-1	10	70%	50%	20%	80%
	6.1-2	20	75%	60%	0%	90%
	6.1-3	24	33%	0%	0%	54%
	6.2-1	12	67%	0%	50%	100%
File Handling	7-1	16	100%	100%	0%	100%
	7-2	18	72%	72%	44%	89%
	7-3	16	25%	50%	0%	56%
	7-4	20	85%	100%	0%	100%
	7-5	20	100%	100%	100%	100%
	7-6	16	81%	69%	0%	100%
Exception Handling	8-1	10	0%	70%	80%	100%
	8-2	24	100%	100%	42%	100%
	8-3	22	100%	0%	36%	100%
	8-4	20	70%	80%	60%	100%
Average Pass Rate			74%	52%	22%	82%
Correct(%)			19(42%)	16(36%)	3(7%)	28(62%)
Wrong Answer(%)			23(51%)	22(49%)	25(56%)	13(29%)
Runtime Errors(%)			3(7%)	7(15%)	17(37%)	4(9%)

After completing the 45 questions in the LLMC dataset, Bing AI's code suggestions for 28 questions are considered correct, resulting in a correct rate of 62%. Bing AI also provide 13 Wrong Answers and 4 answers that could not be executed correctly. These results suggest that Bing AI has a higher correct rate in generating code suggestions for the LLMC dataset, compared to ChatGPT, Claude and Spark.

ChatGPT generates code suggestions for 19 questions that are considered Correct, 23 questions that are considered Wrong Answers, and 3 questions that could not be executed due to Runtime Errors. This is the lowest number of Runtime Errors among the three models. Claude's code suggestions are considered correct for 16 questions, Wrong Answer for 22 questions, and Runtime Error for 7 questions.

Our evaluation of the code generated by Spark indicates that its performance in generating correct code suggestions is not satisfactory. Specifically, out of the 45 questions in the LLMC dataset, Spark generates Correct code suggestions for only 3 questions, while 25 questions are given Wrong Answers, and 17 questions has Running Errors. This poor performance suggests that Spark may have limitations in generating correct code for complex programming tasks.

Q2) **What about the quality of the code given by the large language models?**

Code Complexity. Bing AI has the lowest sum of complexity among the models. Specifically, on the 45 questions of the LLMC dataset, the sum of cyclomatic complexity and cognitive complexity of Bing AI are 209 and 235, respectively. In comparison, the sum of cyclomatic complexity and cognitive complexity of ChatGPT, Claude, and Spark are 220 and 297, 214 and 258, and 243 and 290, respectively. The lower sum of complexity in Bing AI's generated code suggests that it may be more readable and easier to understand than the code generated by the other models.

Code Reliability and ***Code Security***. Evaluation results suggest that the code suggestions from Bing AI, ChatGPT, Claude, and Spark are generally reliable and secure. Only one code suggestion from ChatGPT and one from Spark are found to be buggy during static analysis, resulting in a reliability rating of E.The most of the code suggestions from the four models are rated A for reliability and security, indicating that they are effective in avoiding bugs and vulnerabilities in the code.

Code Readability. Evaluation of the generated code suggests that ChatGPT, Claude, Spark, and Bing AI achieved good average comments rate on the 45 code questions, with rates of 18.4%, 21.7%, 21.9%, and 19.9%, respectively. The comments rate serves as an indicator of code readability, as code with higher comments rate is generally easier to understand. However, there are instances of inaccurate or missing explanations in the code suggestions. Specifically, ChatGPT and Bing AI has fewer instances of inaccurate or missing explanations with 5 and 4 questions, respectively, while Claude and Spark has more with 9 and 16 questions. This suggests that ChatGPT and Bing AI may have stronger code explanation capabilities than Claude and Spark, and may be better suited for generating code that is easy to understand and maintain.

Q3) **What categories of code problems do large language models solve better?**

Our analysis of the generated code suggests that the performance of the four large language models varies depending on the type of code question being considered. Specifically, we count the number of incorrect code suggestions across different types of questions, including those labeled as Wrong Answers and Running Errors, and calculate the percentage of such questions. The results are shown in the Table 6.

It can be observed that ChatGPT, Claude, and Bing AI perform better in dealing with questions related to Data Types and Operators and Control Structure, with an average percentage of problems less than 25%. However, the four models perform more poorly when dealing with questions related to Functions and Modules, Classes and Objects, with more than 85% of code suggestions have problems.

Furthermore, Bing AI shows a clear advantage over the other

Table 6
Incorrect code suggestions ratio of four Large Language Models for different question categories in LLMC Dataset

Questions \ LLMs	ChatGPT	Claude	Spark	Bing AI
Data Types and Operators	17%	33%	100%	0%
Data Structures	73%	64%	91%	45%
Control Structures	33%	33%	67%	33%
Strings and Regular Expressions	43%	71%	100%	43%
Functions and Modules	75%	100%	100%	75%
Classes and Objects	100%	100%	100%	75%
File Handling	67%	50%	83%	33%
Exception Handling	50%	75%	100%	0%

three models in dealing with questions related to File Handling and Exception Handling, with a significantly lower percentage of problems.

Q4) **Are large language models more likely to make understanding problems or code problems?**

We also conduct an analysis of the reasons for the occurrence of incorrect code suggestions generated by the four large language models. We categorize the problems into two categories: code problems and understanding problems. Code problems refer to errors in the code generated by the models, such as using undefined variables or index out of bounds errors, while understanding problems refer to cases where the models do not fully understand the user's requirements, resulting in code suggestions that do not meet the user's expectations. The number of problems in each category is shown in the figure below.

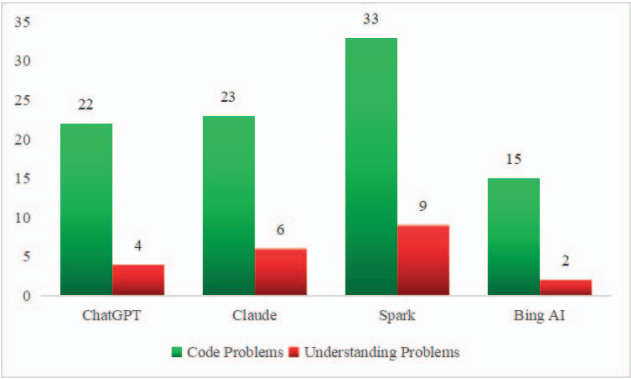

Figure 4. The number of Code Problems and Understanding Problems for different Large Language Models

Our evaluation results suggest that code suggestions generated by the large language models are more likely to have code problems than understanding problems. This implies that the models are better at understanding the user's requirements, but are more likely to make errors in the code generation process. One of the most frequent code problems observed in the evaluation is the failure to import the required packages when writing Python code, which occurred for each of the four large language models. Understanding the reasons for incorrect code suggestions can improve the models' capability to generate correct and efficient code.

5. Conclusions

Our evaluation of the four large language models suggests that their code generation capabilities vary due to differences in architecture, pre-training data, and fine-tuning techniques. Pre-training-based generative models, such as Claude, and GPT-3 and its variants (e.g. ChatGPT), use large amounts of pre-training data to learn the syntax and semantics of programming languages. This enables them to generate more coherent and syntactical correct code, and they show promising results in generating high-quality code.

In contrast, search-based language models, such as Microsoft's Bing AI, employ a different approach by searching for relevant code snippets from the Internet based on user prompts. In our evaluation, Bing AI outperformed the other models in all evaluation dimensions, suggesting that search-based models may have stronger code generation capabilities. This is a possible direction for future development of large language models.

Moreover, our analysis of problem categories indicates that the current large language models have strong natural language understanding abilities. The problems in code suggestions are more likely to be due to code problems rather than understanding problems. Therefore, to improve the code suggestion abilities of large language models, it may be beneficial to focus on improving their ability to generate correct and efficient code.

6. Limitations and Future Work

To demonstrate the effectiveness and rationality of our evaluation method, we test the questions in the LLMC dataset on four large language models, focusing on the Python programming language. As large language models continue to gain popularity, more companies are expected to launch their own large language models in the future. Therefore, it will be important to evaluate the performance of these new models on the LLMC dataset and to investigate whether there are differences in the code generation capabilities of large language models across different programming languages, such as Java and C++.

Moreover, the LLMC dataset currently contains only 45 relatively basic code questions. In future work, we plan to expand the number of questions in the dataset and to increase the difficulty of the questions. This will enable us to evaluate the performance of large language models on more challenging code problems and to identify areas for improvement.

References

[1] J. Wei, Y. Tay, R. Bommasani, C. Raffel, B. Zoph, S. Borgeaud, D. Yogatama, M. Bosma, D. Zhou, D. Metzler, E. H. Chi, T. Hashimoto, O. Vinyals, P. Liang, J. Dean and W. Fedus, "Emergent Abilities of Large Language Models", arXiv preprint arXiv:2206.07682, 2022.

[2] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei and I. Sutskever, "Language Models are Unsupervised Multitask Learners", OpenAI Blog, 2019

[3] J. Ye, X. Chen, N. Xu, C. Zu, Z. Shao, S. Liu, Y. Cui, Z. Zhou, C. Gong, Y. Shen, J. Zhou, S. Chen, T. Gui, Q. Zhang and X. Huang, "A Comprehensive Capability Analysis of GPT-3 and GPT-3.5 Series Models", arXiv preprint arXiv: 2303.10420, 2023.

[4] A. Alzahem, S. Latif, W. Boulila and A. Koubaa, "Unlocking the Potential of Medical Imaging with ChatGPT's Intelligent Diagnostics," arXiv preprint arXiv:2305.07429, 2023.

[5] M. A. Sánchez-Ruiz, J. M. García-Gutiérrez and J. L. Guzmán-Vargas, "Exploring the Protein Sequence Space with Global Generative Models," arXiv preprint arXiv:2305.01941, 2023.

[6] S. Gulwani, O. Polozov, and R. Singh, "Program Synthesis" , Foundations and Trends® in Programming Languages, vol. 4,Now Foundations and Trends, 2017, pp. 112–119

[7] M. Allamanis, E. T. Barr, P. Devanbu and C. Sutton, "A survey of machine learning for big code and naturalness", ACM Computing Surveys (CSUR), vol. 51, no. 4, 2018, pp. 1-37.

[8] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C.J. Cai, M. Terry, Q.V. Le and C. Sutton, "Program Synthesis with Large Language Models", arXiv preprint arXiv:2108.07732, 2021

[9] S. Luan, D. Yang, C. Barnaby, K. Sen and S. Chandra, "Aroma: Code recommendation via structural code search", Proceedings of the ACM on Programming Languages, vol. 3, no. OOPSLA, 2019, pp. 1-28.

[10] GitHub. GitHub Copilot - Your AI pair programmer. 2021.[Online]. Available: https://copilot.github.com/

[11] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. Ponde de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, et al."Evaluating Large Language Models Trained on Code", arXiv preprint arXiv:2107.03374, 2021.

[12] A. Trisovic, M.K. Lau, T. Pasquier and M. Crosas, "A large-scale study on research code quality and execution", Scientific Data, vol. 9, 2021

[13] LeetCode. "Number of Good Ways to Split a String". 2020. [Online]. Available: https://leetcode.com/problems/number-of-good-ways-to-split-a-string/

[14] R. H. Mogavi, X. Ma and P. Hui, "Characterizing Student Engagement Moods for Dropout Prediction in Question Pool Websites", Proceedings of the ACM on Human-Computer Interaction, vol. 5, 2021, pp. 1-22.

[15] G. A. Campbell and P. P. Papapetrou, SonarQube in Action, 1st ed. USA: Manning Publications Co., 2013

[16] C. E. de C. Dantas and M. de A. Maia, "Readability and Understandability Scores for Snippet Assessment: an Exploratory Study", arXiv preprint arXiv:2108.09181, 2021

[17] SonarQube. Metric definitions. 2021. [Online]. Available: https://docs.sonarqube.org/latest/user-guide/metric-definitions/

[18] SonarQube. Cognitive Complexity white paper. 2021. [Online]. Available:https://www.sonarsource.com/resources/cognitive-complexity/

[19] R. Bommasani, D. Soylu, T. I. Liao, K. A. Creel and P. Liang, "Ecosystem Graphs: The Social Footprint of Foundation Models", arXiv preprint arXiv:2303.15772, 2023

[20] iFLYTEC. iFLYTEC Spark. 2023. [Online]. Available: https://xinghuo.xfyun.cn/

[21] 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.-Y. Nie and J.-R. Wen, "A Survey of Large Language Models", arXiv preprint arXiv:2303.18223, 2023