PythonSaga: Redefining the Benchmark to Evaluate Code Generating LLM

Anonymous ACL submission

Abstract

Driven by the surge in code generation using large language models (LLMs), numerous benchmarks have emerged to evaluate these LLMs capabilities. We conducted a large-scale human evaluation of *HumanEval* and *MBPP*, two popular benchmarks for Python code generation, analyzing their diversity and difficulty. Our findings unveil a critical bias towards a limited set of programming concepts, neglecting most of the other concepts entirely. Furthermore, we uncover a worrying prevalence of easy tasks, potentially inflating model performance estimations. To address these limitations, we propose a novel benchmark, `PythonSaga`, featuring 185 hand-crafted prompts on a balanced representation of 38 programming concepts across diverse difficulty levels. The code and dataset are openly available to the NLP community at `https://anonymous.4open.science/r/PythonSaga`.

1 Introduction

The rapid advancement of large language models (LLMs), such as Gemini (Anil et al., 2023a), GPT-4 (OpenAI, 2023), LLaMA (Touvron et al., 2023) and PaLM (Anil et al., 2023b), has achieved near-human or even superhuman performance (Bowman, 2023) across a wide range of NLP tasks. This surge has also prompted the development of tailor-made code generation models, such as Codex (Chen et al., 2021), STARCODER (Li et al., 2023), CodeGen (Nijkamp et al., 2022), and CodeGeeX (Zheng et al., 2023). These specialized models, hereafter collectively referred to as *"Code-LLMs"*, harness the capabilities of LLMs for automated code generation from human descriptions. Figure 1 shows a toy example with an input description from a human and an expected Python code generated by a Code-LLM.

The prevalence of Python as the dominant programming language has significantly influenced a majority of Code-LLMs to showcase their code-generation capabilities on Python-specific benchmarks. Consequently, HumanEval (Chen et al., 2021), MBPP (Austin et al., 2021), APPS (Hendrycks et al., 2021), and DS-1000 (Lai et al., 2023) have emerged as prominent benchmarks, leveraging data curated from popular coding platforms like GitHub (GitHub, 2024), LeetCode (GeeksForGeeks, 2023), and Codeforces (Codeforces, 2024) and crowdsourcing efforts. These benchmarks offer a diverse range of programming challenges, with sizes spanning from a few hundred instances in HumanEval (Chen et al., 2021)) to several thousand instances in datasets like APPS (Hendrycks et al., 2021) and MBPP (Austin et al., 2021).

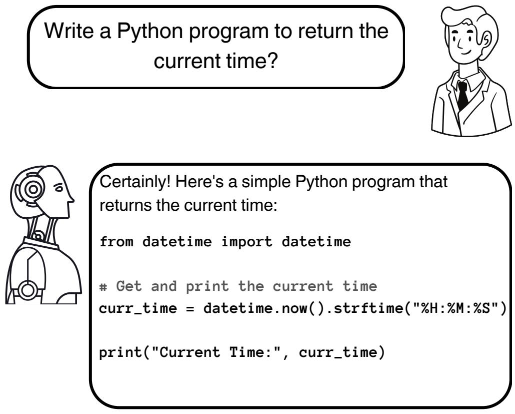

Figure 1: Illustration of a conversation wherein a human provides an input description, and a Code-LLM generates the expected Python code.

Code generation benchmarks, like their NLP counterparts (Kiela et al., 2021), are reaching saturation, revealing limitations in their ability to evaluate models comprehensively. Figure 2 reports *pass@1* score[1] of recent Code-LLMs on two popular benchmarks, HumanEval (Chen et al., 2021)

[1] *pass@k* measures if at least one of the k code samples generated by the model passes every test case. Detailed formal definition is present in Appendix A.1.

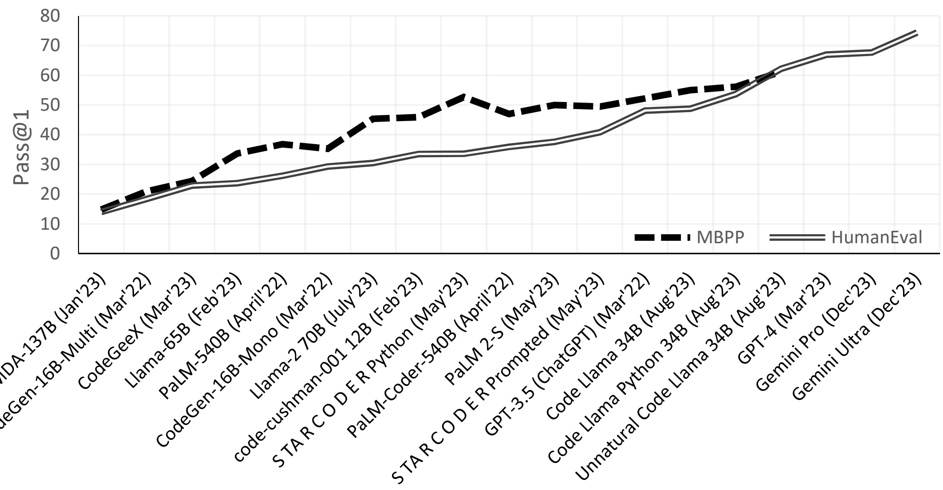

Figure 2: Performance comparison arranged in ascending order of (pass@1) of popular Code-LLMs on two Python benchmarks, HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021). *pass@1* scores are taken verbatim as reported in STARCODER (Li et al., 2023), Code Llama (Roziere et al., 2023), and Gemini (Anil et al., 2023a). GPT-4, Gemini Pro, and Gemini Ultra do not report performance scores on MBPP dataset.

and MBPP (Austin et al., 2021). This progress prompts two critical questions: **(1)** Have Code-LLMs attained the generalization ability to solve any programming problem? **(2)** What programming concepts remain challenging for them, hindering their ability to solve specific problems? Surprisingly, despite their widespread use, existing benchmarks lack a comprehensive evaluation of their diversity in terms of programming concepts and difficulty level.

In this paper, we introduce a comprehensive hierarchical classification of programming concepts, categorizing them into basic, intermediate, and advance levels (see Section 3). We then rigorously evaluate two benchmarks, HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021), on two dimensions: diversity of programming concepts and user-perceived difficulty. Our findings reveal a significant bias towards a small subset (<53%) of programming concepts, leaving the vast majority underrepresented. Additionally, over 80% of the problems are perceived as easy, raising concerns about the benchmarks' generalizability and effectiveness (see Section 4). To address these limitations, in Section 5, we propose a novel code generation benchmark, `PythonSaga`, featuring a balanced representation of 38 programming concepts across three difficulty levels in the form of 185 manually crafted problems. Surprisingly, our experiments show poor *pass@1* scores by the majority of the existing open ($<$ 4%) and closed-source ($<$ 13%) Code-LLMs on `PythonSaga`. Furthermore, detailed analysis unveils significant disparities in their capacity to handle different programming concepts and difficulty levels.

2 Related Work

NLP for Programming: Over the years, various programming tasks, including clone detection (Roy et al., 2009) (assessing semantic similarity between code fragments), defect detection (Tabernik et al., 2020) (identifying potential flaws within source code), code completion (Hindle et al., 2016) (predicting subsequent tokens based on code context), automated code repair (Arcuri and Yao, 2008) (improving code by automatically addressing bugs), code search (Sachdev et al., 2018) (gauging semantic relevance between textual descriptions and code snippets), and code summarization (Allamanis et al., 2016) (generating natural language comments for code), have been extensively investigated and discussed within the NLP community. This exploration has led to the development of several datasets such as GitHub Java Corpus (Allamanis and Sutton, 2013), BigCloneBench (Svajlenko et al., 2014), POJ-104 (Mou et al., 2016), PY150 (Raychev et al., 2016), Devign (Zhou et al., 2019), Bugs2Fix (Tufano et al., 2019), CodeSearchNet (Husain et al., 2019), CT-max/min (Feng et al., 2020), MBPP by Austin et al. (2021), CodeXGLUE by Lu et al. (2021), CodeNet by Puri et al. (2021), HumanEval by Chen et al. (2021), XLCoST by Zhu et al. (2022), MultiPL-E by Cassano et al. (2022), and HumanEval-X by Zheng et al. (2023). These datasets and associated benchmarks span multiple programming languages, including Java, C, C++, PHP, Ruby, Go, and Python, among others.

Code Generation Models: The remarkable surge in the popularity of large language models (LLMs) has also been accompanied by significant advancements in code-generation LLMs (Code-LLMs). These models exhibit the capability to generate code in designated programming languages, guided by instructions presented in the form of prompts, functions, or docstrings. Prominent examples of such Code-LLMs include but are not limited to, Codex (Chen et al., 2021), CodeGen (Nijkamp et al., 2022), Code Llama (Roziere et al., 2023), STARCODER (Li et al., 2023) and CodeGeeX (Zheng et al., 2023). These Code-LLMs are largely multilingual, capable of handling multiple programming languages, and their parameter sizes range from 1 billion to 35 billion. Their training datasets encompass popular programming websites and code repositories such as GitHub, LeetCode, and GeeksForGeeks. All popular Code-LLMs primarily focus on Python programs due to their widespread usage in ML and AI applications.

Python-based Evaluation Benchmarks: Recent thrust in Python code generation models also led to the development of several benchmark datasets. The PY150 dataset (Raychev et al., 2016), consisting of 150,000 Python source files from GitHub, serves as a valuable tool for LLM evaluation. The APPS dataset Hendrycks et al. (2021) features 10,000 problems from platforms like Codewars, AtCoder, Kattis, and Codeforces. HumanEval (Chen et al., 2021) comprises 164 handwritten problems. The MBPP dataset (Austin et al., 2021) contains 974 entry-level problems. Additionally, the MathQA-Python dataset (Austin et al., 2021), with 23,914 problems, assesses code synthesis from complex textual descriptions.

Limitations in Existing Benchmarks: Current datasets for evaluating Large Language Models (LLMs) often lack transparency and comprehensiveness in problem selection and categorization. This opacity hinders assessments of the generalizability and representativeness of the benchmarks, potentially leading to overestimation of LLM performance on code generation tasks. To address this issue, this paper proposes a comprehensive problem categorization by outlining recommended concepts for problem inclusion, aiming to establish a rigorous and transparent benchmarking framework.

3 Programming Concepts and Difficulty Levels

3.1 Programming Concepts

The concepts encompass language-specific constructs like variables, data types, control flow, and conditions to generic constructs like Algorithms, OOPs, etc. We, therefore, propose a hierarchy of programming concepts where a complex concept might require knowledge of several basic concepts. For example, sorting algorithms like `Quicksort` or `Mergesort` require a thorough understanding of data structures such as arrays and linked lists, as well as proficiency in algorithmic analysis and time complexity[2]. Each programming concept is an intrinsic feature of a problem. We next describe the proposed hierarchy:

- **Basic Concepts:** At the basic level, concepts involve the application of elementary syntax principles, encompassing the utilization of variables, manipulation of diverse data types, basic input/output operations, comprehension of control flow and conditional statements, basic handling of data structures, functions, and knowledge of essential built-in libraries. Problems leveraging basic concepts primarily aim to evaluate the core competencies within a designated programming language.
- **Intermediate Concepts**: Intermediate-level concepts involve a comprehensive understanding of multiple foundational concepts and their adept integration. For example, extending basic data structures to implement `Stack`, `Hash`, `Queue`, etc. Problems comprising intermediate concepts evaluate a higher level of proficiency in programming.
- **Advance Concepts:** Concepts include implementation knowledge of advanced data structures such as `Tree`, `Heap`, etc., algorithmic paradigms such as `Greedy`, `Divide and Conquer`, and `Dynamic Programming`, and `Concurrent and Parallel Programming`. Problems comprising advanced concepts focus on evaluating sophisticated problem-solving and design capabilities.

We curate a list of 38 programming concepts from three popular coding platforms (GeeksForGeeks, 2023; LeetCode, 2023; hackerearth,

[2] https://shorturl.at/nrBTX

Basic	Intermediate	Advance
Function	OOPS	Trie
Mathematics	Stack	Tree
File Handling	Sorting	Heap
Basic Libraries	Hashing	Graph
Error Handling	Searching	Matrix
Input and Output	Recursion	Max Flow
In-Built Functions	Linked List	Disjoint Set
Pattern Replication	Bit Manipulation	Backtracking
Basic Data Structures	Queue & Dequeue	Greedy Search
Variable & Data Types	Regular Expression	Advanced OOPs
Control Flow & Conditions	Circular & Doubly Linked List	Context Managers
	Advanced String Manipulation	Divide and Conquer
		Dynamic Programming
		Closures and Decorators
		Concurrency and Parallelism

Table 1: A hierarchy of 38 programming concepts categorized into basic, intermediate, and advance categories.

2023). We further assign each concept to one of the three hierarchy levels. Table 1 presents the curated concepts and the proposed hierarchy.

3.2 Difficulty Levels

An annotator, with their expertise and experience in programming, can perceive a programming problem as belonging to one of three difficulty levels: *Easy*, *Medium*, or *Hard* (Hendrycks et al., 2021). Thus, difficulty level is an extrinsic feature of a problem. This subjective assessment is based on a complex combination of factors, such as knowledge of programming concepts, problem-solving skills, experience with similar problems, and coding proficiency. It is important to note that this perceived difficulty is subjective and can vary significantly between annotators. A problem considered easy by one annotator due to their prior experience and knowledge might be deemed challenging by another who lacks those same advantages. Furthermore, the perceived difficulty of a problem can also evolve over time as an annotator develops their skills and knowledge. A problem that initially seemed challenging may become easier with practice and exposure to similar problems. Conversely, an annotator may encounter a problem that initially appears straightforward but then find themselves struggling due to hidden complexities or unforeseen challenges.

In this paper, we focus on Python Programming language and conduct human experiments with two popular Python-based code generation benchmarks to showcase extensive selection bias and poor diversity in the curated problems. The following section describes the human experiments in detail.

4 Limitations of Code- Generation Existing Benchmarks

4.1 Python Code Generation Benchmarks

This study is grounded on the two most widely recognized Python code generation benchmarks: (i) HumanEval (Chen et al., 2021) and (ii) MBPP (Austin et al., 2021). Recent Code-LLMs including STARCODER (Li et al., 2023), LLaMA (Touvron et al., 2023), METAGPT (Hong et al., 2023), Code Llama (Roziere et al., 2023), SANTACODER (Allal et al., 2023), and CodeGeeX (Zheng et al., 2023) have employed these two benchmarks to assess their performance. We next briefly describe these benchmarks.

- **HumanEval Dataset:** HumanEval dataset was introduced alongside Codex (Chen et al., 2021)[3]. It comprises 164 hand-crafted Python programming problems[4]. Each problem description contains a function signature, docstring, body, and multiple unit tests. Figure 5 illustrates a representative problem. On average, each problem is associated with 7.7 unit tests.

[3] Codex is a GPT-based language model fine-tuned on publicly available codes from GitHub.

[4] Dataset is available here: https://github.com/openai/human-eval

- **Mostly Basic Programming Problems (MBPP) Dataset:** The MBPP dataset (Austin et al., 2021) evaluates models that can synthesize short Python programs from natural language descriptions. The benchmark[5] consists of about 974 crowd-sourced Python programming problems designed to be solvable by entry-level programmers. Each problem consists of a task description, code solution, and three automated test cases. Figure 6 presents a representative problem.

Both benchmarks evaluate model performances against one of the most popular metrics *pass@k*. We formally define *pass@k* in Section A.1.

4.2 Human Annotation Experiments

Next, we conducted two human annotation studies to gain insights into the diversity in programming concepts and difficulty levels of the two proposed benchmarks. Each study involved the recruitment of the same set of five annotators. Each annotator is a postgraduate student in Computer Science with at least three years of experience in Python programming and competitive programming. It is noteworthy that each participant willingly volunteered throughout the entire duration of the experiment, and no remuneration was provided. Internet access was prohibited during the entire annotation period. Annotators were encouraged to utilize any brute-force technique they considered appropriate without prioritizing optimized solutions. No time constraints were imposed to prevent hasty or fatigue-induced decisions. Each annotator was presented with 164 problems from HumanEval and a randomly selected set of 164 problems from MBPP. We next describe the two annotation studies:

- **Programming Concepts Diversity:** In this study, we adopted a single-concept annotation approach, where human annotators assigned one programming concept (detailed in Section 3.1) to each problem. This selection represented the concept they considered most crucial for successful problem-solving. Our annotation guidelines explicitly prohibited assigning multiple concepts to any single problem, ensuring a focused and unambiguous mapping between problems and relevant concepts.
- **Difficulty Level Diversity:** In this study, each annotator categorized the problems into three distinct difficulty levels: *Easy*, *Medium*, and *Hard*, based on their individual expertise and experiences.

[5]Dataset is available here: `https://github.com/google-research/google-research/tree/master/mbpp`

4.3 Observations

Diversity in the Programming Concepts: In this section, we report the proportion of problems assigned to a specific concept averaged over five annotators. We find five predominant concepts, *Mathematics*, *Control Flow and Conditions*, *Basic Data Structures*, *Variables and Data Types*, and *In-Built Functions*, which comprise 72.1% and 77.3% problems in HumanEval and MBPP, respectively. Surprisingly, we found a complete absence of 14 (=37.8%) concepts. Notable exclusions include *OOPs*, *Linked-lists*, *Tree*, *Graph*, and *Backtracking*. Figure 3 presents conceptwise proportions in both the benchmarks. Further analysis suggests that, on average, the Basic category comprises approximately 78% of problems in both HumanEval and MBPP. The Intermediate category comprises 20.24% and 18.04% problems in HumanEval and MBPP, respectively. Finally, the Advance category contains 1.09% and 3.04% problems in HumanEval and MBPP, respectively.

Diversity in the Difficulty level: Here, we report the difficulty level assigned to a problem using majority voting among the annotators. In HumanEval, 84.8% of the problems were classified as *Easy*, 14.6% as *Medium*, and only 0.6% as *Hard*. Whereas in MBPP, 89.6% and 10.4% of problems were categorized as *Easy*, and *Medium*, respectively. No problem in MBPP was labeled as *Hard*. Here, we achieved significant consensus among the annotators. For example, in HumanEval, we find complete agreement among five annotators on 39% of the problems. Whereas we miss complete agreement by a single vote in 29.2% problems. In the case of MBPP, the 40.2% problems resulted in a complete agreement, with 42.1% problems missing the complete agreement by one vote.

Overall, we observe significant selection bias towards easy problems in both benchmarks.

5 `PythonSaga`: A New Benchmark for Code Generation Models

We now introduce `PythonSaga`, a new Python code generation benchmark that addresses the limitations of existing benchmarks with respect to diversity in concepts and difficulty level. `PythonSaga`

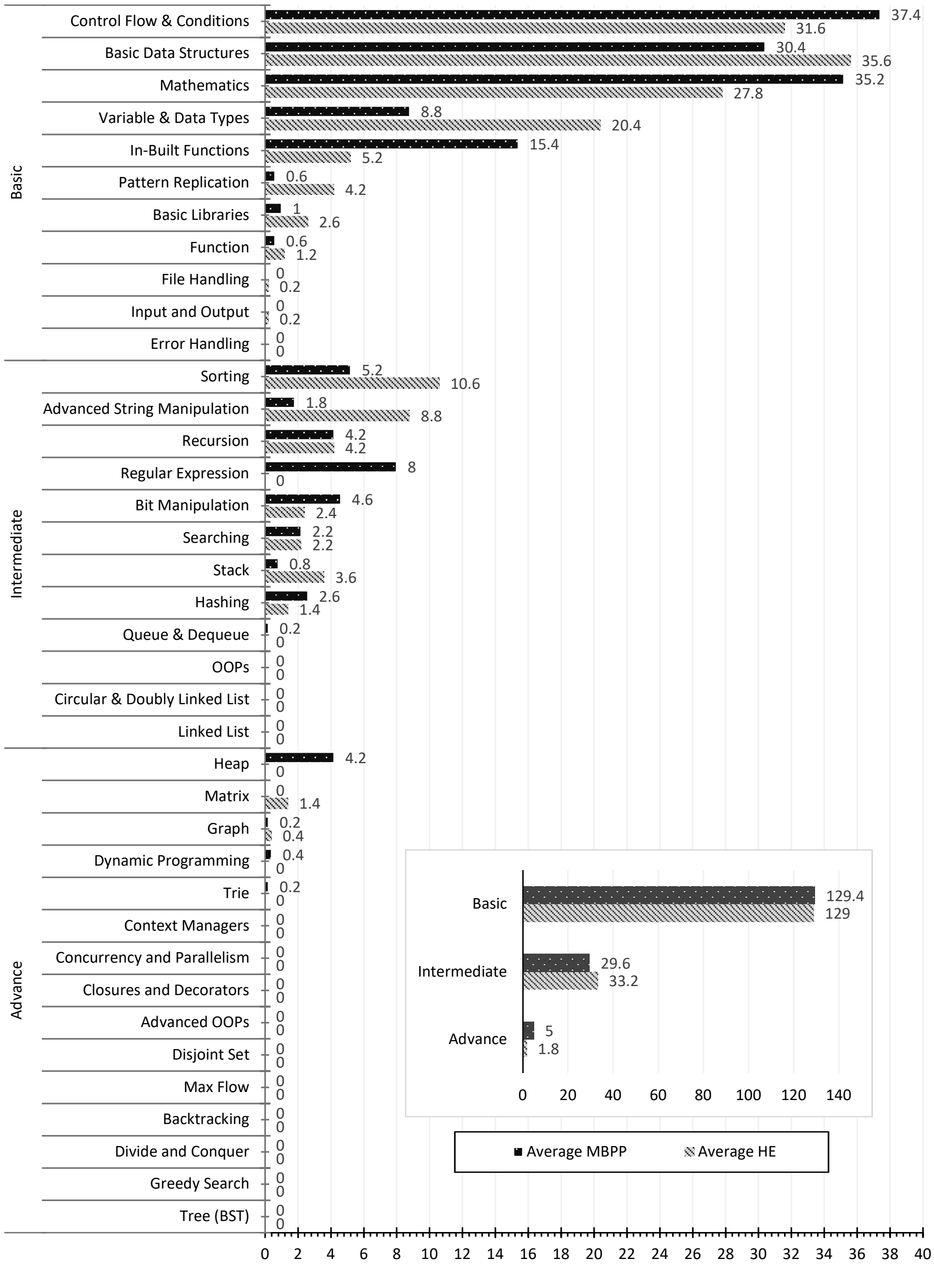

Figure 3: Average number of problems/prompts in Fine-Grain category

contains 185 prompts, close to equal representation from each of the 38 programming concepts with varied levels of difficulty (described in Section 3.2).

5.1 Data Sources and Curation Methodology

Aligned with the problem curation strategies employed in established benchmarks Hendrycks et al. (2021); Lai et al. (2023); Zhu et al. (2022), this study leverages problems from two prominent coding platforms: GeekForGeeks (GeeksForGeeks, 2023) and LeetCode (LeetCode, 2023). To comprehensively represent each proposed programming concept (detailed in Section 3.1), we curated five problems per concept. This diverse set comprises one *Easy* problem, two *Medium* problems, and two *Hard* problems, ensuring a balanced distribution across difficulty levels (20%, 40%, and 40%, respectively) within the `PythonSaga` Dataset.

To enhance human-friendliness and ground the problems in realistic contexts, each shortlisted problem statement undergoes a manual rephrasing process without any aid from AI tools. Furthermore, a comprehensive description of input and output formats, accompanied by relevant examples, is supplied with each problem statement to ensure a thor-

Model	Size	Pass@1	Pass@10
Code Llama Python (Roziere et al., 2023)	7B	0.0240	0.0979
Code Llama Instruct (Roziere et al., 2023)	7B	0.0178	0.0744
Mistral-Instruct-v0.1 (Jiang et al., 2023)	7B	0.0140	0.0552
Code Llama (Roziere et al., 2023)	7B	0.0067	0.0472
StarCoderBase (Li et al., 2023)	7B	0.0029	0.0149
Deepseek Coder Instruct (Guo et al., 2024)	6.7B	0.0137	0.0889
Deepseek Coder (Guo et al., 2024)	6.7B	0.0343	0.1415
GPT-3.5 (OpenAI, 2022)	NA	0.0724	0.2384
GPT-4 (OpenAI, 2023)	NA	0.1243	0.3311

Table 2: Comparison between open and closed-source models on PythonSaga. We use the number of samples (n) as 20 for both open and closed-source models. OpenAI has not officially released the sizes of GPT-3.5 and GPT-4.

ough understanding of the task by Code-LLM. This multi-step approach aims to retain the core knowledge and essential solution steps while integrating them into relatable real-world scenarios. This reconstruction involves reformulating the entire problem statement while preserving its fundamental functionality. This deliberate transformation enhances the challenge for Code-LLMs, requiring them to move beyond simple pattern matching and grasp the nuanced context embedded within the prompt to devise a solution effectively. For example, the problem statement *"Given a string str, find the length of the longest substring without repeating characters."* is paraphrased as *"Let's say you attend a car show where cars of different brands are showcased in a row. Find the length of the longest stretch where no two cars are of the same brand. Take the input from the user for the brands of the cars in the order they are placed in the row. Print the length of the longest stretch where no two cars are of the same brand"*.

5.2 Size and Structure

Overall, PythonSaga comprises five problem instances from each programming concept, resulting in a total size of 185 problems. Each problem is associated with a maximum of four test cases, with an average of 3.7 test cases per problem. PythonSaga's structure resembles HumanEval and MBPP, wherein each problem comprises a function signature, docstring, body, and multiple unit tests. A representative example is present in Appendix A.2.

5.3 Benchmarking Existing LLMs

Next, we benchmark several open and closed-source LLMs on PythonSaga. Open-source models include three Llama variants, Code Llama (Roziere et al., 2023), Code Llama Python (Roziere et al., 2023) and Code Llama Instruct (Roziere et al., 2023), Mistral-Instruct (Jiang et al., 2023), StarCoderBase (Li et al., 2023) and two Deepseek variants, Deepseek Coder (Guo et al., 2024) and Deepseek Coder Instruct (Guo et al., 2024). Except for Mistral-Instruct, the rest are the Code-LLMs. In addition, we benchmark two closed-source models, including GPT variants GPT-3.5 (OpenAI, 2022) and GPT-4 (OpenAI, 2023). While larger open-source options exist, our selection was restricted to models with 7B parameters due to computational resource limitations, which were limited to a single Tesla V100 in our case.

We evaluate model performances using *pass@k* metric. Adhering to previous studies like HumanEval (Chen et al., 2021), StarCoder (Li et al., 2023), Deepseek Coder (Guo et al., 2024) etc, we primarily utilized $k = 1$, signifying that a model is considered successful if at least one of its generated solutions passes the defined evaluation criteria. However, we additionally explored $k = 10$ to analyze model consistency across larger sets of responses. Notably, unlike prior works that varied the number of sampled responses (n), we consistently generated $n = 20$ samples from both open-source and Closed-source models for a consistent evaluation.

Table 2 compares the above models against *pass@1* and *pass@10* metrics. In consistent with the latest trends, closed-source models performed considerably better than open-source models. Among open-source models, Deepseek Coder (Guo et al., 2024) performed best, whereas, among closed-source models, GPT-4 (OpenAI, 2023) performed best. Notably, the performance

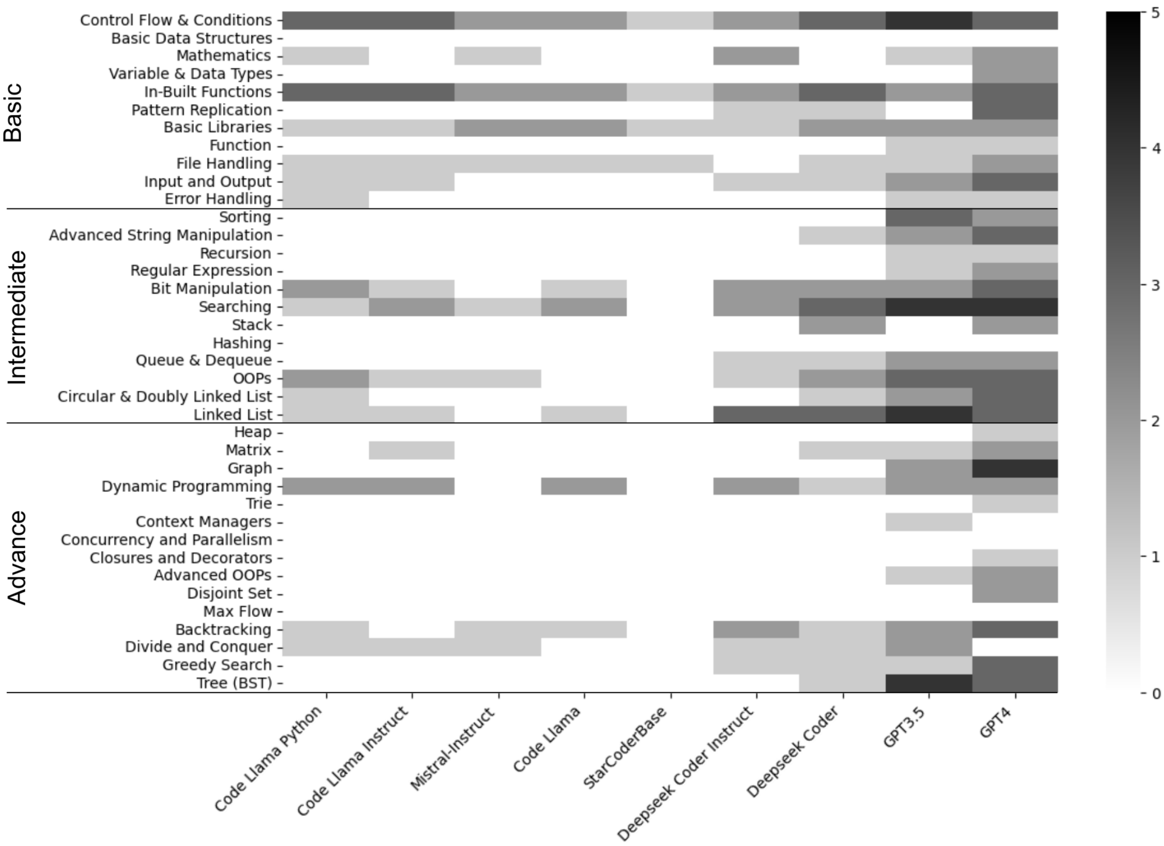

Figure 4: A heatmap showing the number of problems in `PythonSaga` solved by each LLM for a given programming concept. A model succeeds if at least one of the n(=20) generated samples passes all test cases.

of closed-source models on `PythonSaga` is significantly lower than the respective performances in HumanEval and MBPP benchmarks (see Figure 2 for more details).

Figure 4 illustrates the performance of each LLM on problems within specific programming concepts in the `PythonSaga`. We consider a model has successfully solved a problem if any one of the n(=20) generated samples passes all the test cases. As anticipated, all models exhibited better performance in solving problems associated with basic concepts compared to intermediate or Advance concepts. For example, Deepseek Coder, solved 21.1%, 25%, and 8.2% of problems in these categories, respectively. Whereas, GPT-4 solved 42.3%, 46.6%, and 32.8% of problems, respectively. In contrast to open-source models, closed-source models have successfully solved at least one problem from a majority of the concepts. Interestingly, none of the models could successfully solve any problem within five specific concepts, *Basic Data Structures*, *Hashing*, *Context Managers*, *Concurrancy and Parallelism* and *Max Flow*. Notably, closed-source models exhibited a more consistent performance across categorization compared to open-source models, suggesting a potential advantage in handling diverse problem complexities.

6 Conclusion and Future Work

This study emphasizes the crucial need for a more balanced and comprehensive evaluation framework to ensure a fair and accurate assessment of large language models (LLMs) capable of generating code from human inputs. We address this gap by proposing an extensive categorization and hierarchy of programming concepts. Subsequent analysis of two prominent Python code generation benchmarks reveals limited diversity in both programming concepts and difficulty levels. Notably, we introduce a novel benchmark characterized by a uniform representation of concepts and difficulty, offering a more robust assessment paradigm. Our findings suggest that existing benchmarks potentially overestimate LLM performance on code generation tasks. This work lays the groundwork for the future development of diverse and representative Python code generation benchmarks, paving the way for similar studies in other programming languages.

Limitations

This section acknowledges three key limitations associated with the present research. Firstly, due to constraints in human annotation resources, the study employed a randomly selected subset of 164 problems from the MBPP benchmark. This selection aimed to match the size of the HumanEval dataset for comparative analysis. While maintaining parity in dataset size was crucial, it is important to acknowledge that the study's findings may not generalize to the entire MBPP benchmark due to the potential for selection bias introduced by the random sampling process. Secondly, the current study employed a team of postgraduate Computer Science students with extensive experience in Python programming and competitive coding. While this selection ensured a high level of technical proficiency in the annotation task, it also acknowledges the potential limitations in terms of annotator diversity. Lastly, while the current study demonstrates the efficacy of our proposed approach within the context of the Python programming language, the generalizability of these findings to other languages requires further investigation, potentially limiting the direct applicability of our findings to benchmarks designed for languages such as Java or C++.

Ethics Statement

All human participants engaged in the evaluation process received detailed and comprehensible information regarding the study's nature and objectives. Prior to their involvement in the research, explicit informed consent was obtained from each participant.

References

Loubna Ben Allal, Raymond Li, Denis Kocetkov, Chenghao Mou, Christopher Akiki, Carlos Munoz Ferrandis, Niklas Muennighoff, Mayank Mishra, Alex Gu, Manan Dey, et al. 2023. Santacoder: don't reach for the stars! *arXiv preprint arXiv:2301.03988*.

Miltiadis Allamanis, Hao Peng, and Charles Sutton. 2016. A convolutional attention network for extreme summarization of source code. In *International conference on machine learning*, pages 2091–2100. PMLR.

Miltiadis Allamanis and Charles Sutton. 2013. Mining source code repositories at massive scale using language modeling. In *2013 10th working conference on mining software repositories (MSR)*, pages 207–216. IEEE.

Rohan Anil, Sebastian Borgeaud, Yonghui Wu, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M Dai, Anja Hauth, et al. 2023a. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805*.

Rohan Anil, Andrew M Dai, Orhan Firat, Melvin Johnson, Dmitry Lepikhin, Alexandre Passos, Siamak Shakeri, Emanuel Taropa, Paige Bailey, Zhifeng Chen, et al. 2023b. Palm 2 technical report. *arXiv preprint arXiv:2305.10403*.

Andrea Arcuri and Xin Yao. 2008. A novel co-evolutionary approach to automatic software bug fixing. In *2008 IEEE Congress on Evolutionary Computation (IEEE World Congress on Computational Intelligence)*, pages 162–168. IEEE.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.

Samuel R Bowman. 2023. Eight things to know about large language models. *arXiv preprint arXiv:2304.00612*.

Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q Feldman, et al. 2022. Multipl-e: A scalable and extensible approach to benchmarking neural code generation. *arXiv preprint arXiv:2208.08227*.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Codeforces. 2024. Codeforces.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*.

GeeksForGeeks. 2023. Gfg.

GitHub. 2024. Github.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y Wu, YK Li, et al. 2024. Deepseek-coder: When the large language model meets programming–the rise of code intelligence. *arXiv preprint arXiv:2401.14196*.

hackerearth. 2023. hackerearth.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, et al. 2021. Measuring coding challenge competence with apps. *arXiv preprint arXiv:2105.09938*.

Abram Hindle, Earl T Barr, Mark Gabel, Zhendong Su, and Premkumar Devanbu. 2016. On the naturalness of software. *Communications of the ACM*, 59(5):122–131.

Sirui Hong, Xiawu Zheng, Jonathan Chen, Yuheng Cheng, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, et al. 2023. Metagpt: Meta programming for multi-agent collaborative framework. *arXiv preprint arXiv:2308.00352*.

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*.

Albert Q Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, et al. 2023. Mistral 7b. *arXiv preprint arXiv:2310.06825*.

Douwe Kiela, Max Bartolo, Yixin Nie, Divyansh Kaushik, Atticus Geiger, Zhengxuan Wu, Bertie Vidgen, Grusha Prasad, Amanpreet Singh, Pratik Ringshia, et al. 2021. Dynabench: Rethinking benchmarking in nlp. *arXiv preprint arXiv:2104.14337*.

Sumith Kulal, Panupong Pasupat, Kartik Chandra, Mina Lee, Oded Padon, Alex Aiken, and Percy S Liang. 2019. Spoc: Search-based pseudocode to code. *Advances in Neural Information Processing Systems*, 32.

Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Wen-tau Yih, Daniel Fried, Sida Wang, and Tao Yu. 2023. Ds-1000: A natural and reliable benchmark for data science code generation. In *International Conference on Machine Learning*, pages 18319–18345. PMLR.

LeetCode. 2023. Leetcode.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. 2023. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664*.

Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. 2016. Convolutional neural networks over tree structures for programming language processing. In *Proceedings of the AAAI conference on artificial intelligence*, volume 30.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474*.

OpenAI. 2022. Introducing chatgpt. https://openai.com/blog/chatgpt.

OpenAI. 2023. Gpt-4 technical report.

Ruchir Puri, David S Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladimir Zolotov, Julian Dolby, Jie Chen, Mihir Choudhury, Lindsey Decker, et al. 2021. Codenet: A large-scale ai for code dataset for learning a diversity of coding tasks. *arXiv preprint arXiv:2105.12655*.

Veselin Raychev, Pavol Bielik, and Martin Vechev. 2016. Probabilistic model for code with decision trees. *ACM SIGPLAN Notices*, 51(10):731–747.

Chanchal K Roy, James R Cordy, and Rainer Koschke. 2009. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of computer programming*, 74(7):470–495.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

Saksham Sachdev, Hongyu Li, Sifei Luan, Seohyun Kim, Koushik Sen, and Satish Chandra. 2018. Retrieval on source code: a neural code search. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, pages 31–41.

Jeffrey Svajlenko, Judith F Islam, Iman Keivanloo, Chanchal K Roy, and Mohammad Mamun Mia. 2014. Towards a big data curated benchmark of inter-project code clones. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 476–480. IEEE.

Domen Tabernik, Samo Šela, Jure Skvarč, and Danijel Skočaj. 2020. Segmentation-based deep-learning approach for surface-defect detection. *Journal of Intelligent Manufacturing*, 31(3):759–776.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*.

Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2019. An empirical study on learning bug-fixing patches in the wild via neural machine translation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 28(4):1–29.

Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Zihan Wang, Lei Shen, Andi Wang, Yang Li, et al. 2023. Codegeex: A pre-trained model for code generation with multilingual evaluations on humaneval-x. *arXiv preprint arXiv:2303.17568*.

Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *Advances in neural information processing systems*, 32.

Ming Zhu, Aneesh Jain, Karthik Suresh, Roshan Ravindran, Sindhu Tipirneni, and Chandan K Reddy. 2022. Xlcost: A benchmark dataset for cross-lingual code intelligence. *arXiv preprint arXiv:2206.08474*.

A Appendix

A.1 Performance Evaluation

Within the field of code-generating large language models (Code-LLMs), the *pass*@k metric has emerged as a prevalent benchmark for performance evaluation (Kulal et al., 2019). This metric quantifies the overall proportion of benchmark problems successfully solved by a given model. A problem is considered solved if at least one of the k code samples generated by the model passes every test case associated with the problem. However, this definition leads to high variance. HumanEval (Chen et al., 2021) proposed an unbiased variant, where they generate n samples per problem such that n $\geq$ k, and count the number of correct samples $c \leq n$ which pass unit tests. The unbiased estimator is described as:

$$\text{pass@}k := \mathop{\mathbb{E}}_{\text{Problems}} \left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}} \right] \tag{1}$$

Most of the Code-LLMs report *pass*@k values at $k = 1$. However, the value of n varies significantly across models. For instance, STARCODER (Li et al., 2023) conducts experiments with $n = 200$ for open-source models and $n = 20$ for API models.

A.2 Representative Example from PythonSaga

```
{
    "task_id": "PythonSaga/15",

    "prompt":

    def toy_distribution(n: int) -> str:
        """
        Let's say I have a bag of toys,
        which are 'n' in number. I know
        that these toys can be
        distributed either to n children
        or 1 child.
        I want to know what can be other
        ways to distribute these toys to
        children in such a way that each
        child gets at least an equal
        number of toys.
        Take input from the user
‘       for the number of toys. Use the
        divmod function to solve this
        problem.

        Example 1:
        If 15 toys are there, then 15
        children can get 1 toy each or 1
        child can get 15 toys or 3
        children can get 5 toys each or
        5 children can get 3 toys each.
        In this case,
        return 'Yes, it is possible'.

        Example 2:
        If 11 toys are there, then 11
        children can get 1 toy each or
        1 child can get 11 toys.
        In this case,
        return 'No, it is not possible'.
        """,

    "entry_point": "toy_distribution",

    "canonical_solution":
    def is_prime(n):
        """
        Check if a number is prime using
        divmod.
        """
        if n < 2:
            return False

        for i in range(2,int(n**0.5)+1):
            quot,remainder=divmod(n,i)
            if remainder == 0:
                return False

        return True

    def toy_distribution(n: int) -> str:
        if n <= 0 or not is_prime(n):
            return 'Yes, it is possible'

        return 'No, it is not possible',

    "test":
    METADATA = {
        'author': 'AY',
        'dataset': 'test'
    }
    def check(candidate):
        assert candidate(15) == 'Yes,
                                it is possible'
        assert candidate(11) == 'No,
                            it is not possible'
        assert candidate(20) == 'Yes,
                                it is possible'
        assert candidate(2) == 'No,
                             it is possible'
}
```

A.3 Representative Example from HumanEval

```
{
    "task_id":"HumanEval/23",

    "prompt":
        """
        def strlen(string: str) -> int:
            Return length of given
            string
            >>> strlen('')
                0
            >>> strlen('abc')
                3""",

    "entry_point": "strlen",

    "canonical_solution":
        "return len(string)",

    "test":
        """METADATA = {
            'author': 'jt',
            'dataset': 'test'
            }
        def check(candidate):
           assert candidate('') == 0
           assert candidate('x') == 1
           assert candidate('asdasnakj')
                                == 9"""

}
```

Figure 5: Representative example from the HumanEval dataset. Here, *task_id* is a unique identifier for the data sample. The *prompt* contains problem text with a function header and docstrings. *Canonical_solution* presents one solution for the problem. The *test* contains functions to validate the correctness of the generated code. *Entry_point* represents the function name which is yet to be completed.

A.4 Representative Example from MBPP

```
{
    "text": "Write a function to find m
    number of multiples of n.",

    "code":
        '''
        def multiples_of_num(m,n):
           multiples_of_num=
               list(range(n,(m+1)*n,n))
           return list(multiples_of_num)
        ''',

    "task_id": 21,

    "test_setup_code": "",

    "test_list":
        '''
        ["assert multiples_of_num(4,3)==
            [3,6,9,12]",
         "assert multiples_of_num(2,5)==
            [5,10]",
         "assert multiples_of_num(9,2)==
            [2,4,6,8,10,12,14,16,18]"]
         ''',

    "challenge_test_list": []
}
```

Figure 6: Representative example from the MBPP dataset. *Text* represents the natural language description of the problem. *Code* contains one possible solution for the problem. *Task_id* is the unique identifier of the sample. *Test_setup_code* lists necessary code imports to execute tests. *Test_list* contains a list of tests to verify the solution. *Challenge_test_list* contains a list of more challenging tests to probe the solution further.