On Leakage of Code Generation Evaluation Datasets

Alexandre Matton, Tom Sherborne, Dennis Aumiller,
Elena Tommasone, Milad Alizadeh, Jingyi He,
Raymond Ma, Maxime Voisin, Ellen Gilsenan-McMahon
Matthias Gallé
Cohere

Abstract

In this paper, we consider contamination by code generation test sets, in particular in their use in modern large language models. We discuss three possible sources of such contamination and show findings supporting each of them: (i) direct data leakage, (ii) indirect data leakage through the use of synthetic data and (iii) overfitting to evaluation sets during model selection. To address this, we release LBPP: an uncontaminated new benchmark of 161 prompts with their associated Python solutions. LBPP is released at `https://huggingface.co/datasets/CohereForAI/lbpp`.

1 Introduction

Code generation has emerged as an important skill for large language models to master. Measuring recent progress in code generation has relied on few, critical benchmarks to judge performance between model families and checkpoints. While many recent sophisticated evaluation datasets have been proposed (Jain et al., 2024; Jimenez et al., 2024), the community largely relies on `HumanEval` (Chen et al., 2021) and `MBPP` (Austin et al., 2021) to judge a new model's code capability. In fact, all major announcements in 2023-2024 claiming advanced code capabilities—from either academic or industry labs—boast one or both of these datasets. Practically, reporting `HumanEval` and `MBPP` is mandatory for a model to report competitive code generation.

However, the importance of these benchmarks has led to a conflict between popularity and utility. Obtaining competitive numbers comes with significant scientific and economic reward—made increasingly easy with the proliferation of public replicas and evaluation harnesses featuring these datasets. However, this prevalence has led to data leakage beyond the original evaluation scope. When this evaluation data *contaminates* model training, the validity of the metrics as a measure of generalization capability becomes unreliable. If a model is trained on data used for out-of-distribution generalization (or selected for its performance on that data), we break an implicit tenet of how model capability should be measured. We argue that understanding the effect of contamination is critical to accurately interpreting scores on these benchmarks.

In this paper, we review the evidence that most contemporary LLMs are *contaminated* with data sourced from these two benchmarks. We define *contamination* here as any procedure leaking datasets *during* model training such that these datasets are no longer unseen at inference. The most obvious method of contamination is the presence inside training data. Section 3.1 reviews evidence that these benchmarks are widespread in training corpora in original and paraphrased forms. Unfortunately, it is not feasible to manually remove all the corresponding examples from the training corpora and the most common automatic decontamination methods have low recall. Section 3.2 proposes that contamination also occurs indirectly through the use of synthetic data—a widespread paradigm used to increase coding capabilities by generating additional code training tokens for pretraining or fine-tuning. Finally, Section 3.3 argues that checkpoint selection may be overly influenced by these datasets, overfitting to these benchmarks over general-purpose code-oriented generalization.

In this paper, we propose a more challenging Python code generation benchmark: **Less Basic Python Problems**. LBPP is similar in size to `HumanEval` and `MBPP`, but designed to be more complex using model in the loop filtering. LBPP is also designed to share no inspiration or sources with existing training and evaluation datasets, presenting a novel generalization challenge to contemporary LLMs. In Section 4, we observe that SOTA models on `HumanEval` perform up to 43% worse on LBPP. We contribute LBPP as a genuinely held-out test to measure *current* code generation capability, and potential overfitting to `HumanEval` and `MBPP`.

Findings of the Association for Computational Linguistics: EACL 2024, pages 13215–13223
November 12-16, 2024 ©2024 Association for Computational Linguistics

2 Related Work

`HumanEval` (Chen et al., 2021) and `MBPP` (Austin et al., 2021) remain the most reported results on public leaderboards, but similar datasets exist (Hendrycks et al., 2021; Li et al., 2022). They consist of short and mostly simple (not programming competition level) instructions with completions in Python. These datasets have also been translated into more programming languages (Muennighoff et al., 2023; Cassano et al., 2022), as well as versions with additional tests (Liu et al., 2024b).

Jain et al. (2024) proposes a continuously updated set of interview questions to improve dataset challenge by including harder and novel (unseen) prompts. Jimenez et al. (2024) aims for challenging software engineering problems requiring understanding of entire repositories. In a similar vein, RepoQA (Liu et al., 2024a) and Bug In The Code Stack (Lee et al., 2024) focus on understanding long contexts within code tasks. Zhang et al. (2024) proposes using hidden evaluation sets, however, this approach does not allow inspection of failure cases and requires trusting the quality and correctness of an opaque 'black-box' evaluation setup.

Recently, Riddell et al. (2024) analyzed data contamination in popular pretraining datasets: reporting that 12.2% of `HumanEval` samples are present in The Pile (Gao et al., 2020), and 18.9% in The Stack (Kocetkov et al., 2022). While Riddell et al. (2024) reports "we do not find the performance rankings of the models to change with decontaminated results", we identify the ranking between models to vary between contaminated and uncontaminated evaluation datasets (see Table 2).

3 Possible sources of contamination

We provide three hypotheses, with respective evidence, on why existing evaluation datasets are leaked and models may already be over-optimized towards existing leaked benchmarks.

3.1 Direct data leakage

The most obvious reason is the simplest: many of the test datasets are of widespread use and the simplest answer is that modern LLMs are trained on this evaluation data. We note that intentional (i.e., to *cheat*) or unintentional contamination has the same net effect: training on evaluation data limits the confidence and utility of the benchmark.

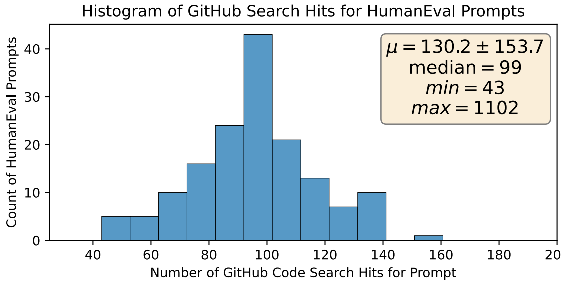


Figure 1: Histogram (excluding outliers) of occurrences for `HumanEval` prompts in public GitHub repositories. Every prompt occurs at least 43 times.

Curating high-quality datasets of natural language to code instructions can incur exorbitant costs when one example may cost upwards of dozens of US dollars. For any party considering the Pareto optimality of dataset size and coding performance gain, the required funding to create novel data can quickly explode. This leads to a common practice of web scraping code-oriented resources (e.g., GitHub or Stack Overflow) for data. However, these resources are also likely sources of contamination. The small data size and portability of such benchmarks encourages replication. We demonstrate this proliferation by keyword searching for `HumanEval` prompts on GitHub. Fig. 1 shows that we return a hit in all cases—the median hits is 99 and the minimum 43. These hits are often exact duplicates indicating a fork of the original dataset.

While decontamination of training sets is becoming more common, present decontamination filters designed for natural text adapt poorly to code. To operate efficiently at scale, most filters rely on generic deduplication algorithms e.g., such as n-gram matching or hashing functions (Lee et al., 2022). Such surface-level matching does not adequately capture code similarity where a simple variable name change leaves program semantics unchanged, but changing a single keyword can have profound changes.[1] As an example, Elazar et al. (2024) report that only 1.22% of verbatim `HumanEval` is present in the OSCAR popular web corpus. The same shortcomings of decontamination apply to the creation of large-scale synthetic datasets: for example, the model-generated dataset of Starcoder (Li et al., 2023) is decontaminated only by removing exact docstrings or solutions that match `HumanEval` or `MBPP`.

The recent exploration of Riddell et al. (2024) quantifies the proportion of this data leakage in

---

[1] Compare the instruction "`return true if string is float`" with "`return true if string is verb`".

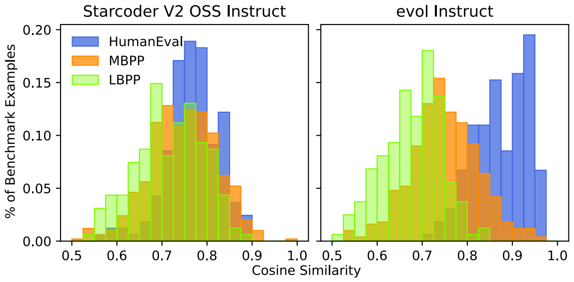


Figure 2: Histogram of cosine similarities for prompts in `HumanEval`, `MBPP` and LBPP relative to two popular synthetic code training datasets. We note the high similarity between most `HumanEval` prompts to *evol-instruct*, and how LBPP has reduced overall similarity to either training dataset.

existing datasets using plagiarism tools specifically designed for code. Even when static training datasets are cleaned, contamination may persist through incremental leakage in other sources. For example, entities serving models via an API may encounter these benchmark examples when evaluated by third-party users. When a sample of real model usage is annotated for future training data, samples from benchmark evaluation can leak into future training corpora. Furthermore, these samples may include paraphrases and format changes that further complicate heuristic deduplication. In this scenario, a model may easily memorize completions to purportedly novel prompts. As evidence of this phenomenon, we prompted one popular commercial system (kept anonymous) with partial prompts from `HumanEval` that were designed to keep the instruction under-specified. Table 4 in Appendix A shows the outcome and evidence that—despite the ambiguity of the prompt—the result matches exactly the gold solution from `HumanEval`.

3.2 Data leakage through synthetic data

The most capable of code language models rely heavily on the use of synthetic training data (Xu et al., 2023; Wei et al., 2023, 2024; Llama Team, 2024). A typical pipeline consists of curating prompts related to code generation, inferring completions with a previously trained LLM, and synthesizing unit tests for relevant prompts using LLMs. Completions passing the respective unit tests are considered valid code solutions and can be used as future training examples. Alternatively, if a sufficiently powerful model is used, completions might be used as-is without validation.

Synthetic data unlocks scales that are usually not reachable with human-labeled data. Common synthetic code datasets generally have between tens of thousands and millions of examples. For example, *Starcoder2-Instruct* is a code dataset of around 238k instances (prior to deduplication) that was created by sampling code from GitHub and using it as seeds to generate self-contained code problems, solutions, and associated tests. `evol-instruct` is another widely popular dataset used by code LLMs such as WizardCoder (Xu et al., 2023) for training. It comprises $110k$ complex query prompts with non-verified completions from closed and open-source models.[2] The sheer size of these datasets – compared to the domain they are targeting – might explain some memorization. After all, the number of unique, self-contained interview-like prompts with a reasonable size is fairly limited, and it is possible that synthetic datasets cover a majority of this space. When training data covers most examples of a domain, it does not matter whether the model memorizes the training data or whether it can generalize further. Table 3 shows some examples of very similar (but not necessarily equivalent) data between `evol-instruct` and `MBPP`.

However, the use of a synthetic data pipeline might hide real leakage. Prior reports (Yu et al., 2023, page 8), (Wei et al., 2023, page 4) discuss an apparent high similarity between examples in `evol-instruct` and `HumanEval`. We also found a lot of semantically equivalent prompts between these two datasets and displayed some in Table 5. We extend this analysis by studying the similarity between *embedded representations*[3] of the prompts of `HumanEval` and `MBPP` with nearest neighbors from `evol-instruct` and *Starcoder-Instruct*. Fig. 2 highlights a widespread similarity between synthetic training datasets and public evaluation data, while the similarity with LBPP is uniformly lower. This is despite LBPP having a very similar format to `MBPP` and `HumanEval` (short prompts asking to solve logic problems). The main difference between `MBPP`/ `HumanEval` and LBPP is the difficulty level: LBPP's questions are generally harder (see Section 4). This inference aligns with observations when fine-tuning a 'Command R Refresh' model adding `evol-instruct`. `HumanEval` score increases by +9%, `MBPP` increases by +2%, but the LBPP score is unchanged.

Whether the high similarity of synthetic datasets with public evaluation data is due to synthetic data

---

[2] Per downloads, the most popular version is a 'lightly decontaminated' version on HuggingFace here.

[3] Embedded using Cohere `embed v3` (Team, 2024).

filling the space of problems similar to `HumanEval`/ `MBPP` or more direct leakage (eg, through the use of in-context examples), these results point to a larger issue. `HumanEval`/ `MBPP` cannot be used as the only proxies to evaluate a model's code abilities. They mostly provide code performance signal on a very specific type of problems with a very specific level of difficulty. We need more diversity in the code evaluation benchmarks and we believe that LBPP is a step in the right direction.

3.3 Overfitting to test sets

The exaggerated importance of a few benchmarks encourages an incentive structure where model selection prioritizes gain on a narrow suite of metrics. While it may be tempting to use such benchmarks as a deciding factor between similar checkpoints, section 3.2 shows that the correlation between these benchmarks and 'solving code generation' is weak. While the meaning and measurement of this unscientific objective are subject to constant revision, selecting for optimal `HumanEval` performance is akin to p-hacking in other fields. This can be justified by assuming these benchmarks are the new dev sets, while the true test is the usage of users over time. However, the usefulness of a dev set entirely relies on its similarity with the actual use case.

Moreover, some risk remains that models overfit to these 'lucrative' benchmarks, distorting the perception of downstream performance. Table 2 and Fig. 3 illustrate this problem well. Even though the correlation between `MBPP`/`HumanEval` scores and LBPP scores is strong, some models are ranking noticeably higher on `MBPP`/`HumanEval` than on LBPP.

The ultimate effect is imbalanced optimization solely towards these metrics, further motivating the practices outlined in Sections 3.1 and 3.2.

4 LBPP: Less Basic Python Problems

As mentioned above, we have created **Less Basic Python Problems (LBPP)** for a less biased measure of modern models' code capabilities. LBPP is a dataset of 161 code completion problems in the style of `HumanEval`. All prompts include evaluation unit tests, with a median of 4 tests per example.

Dataset Annotation: Human annotators were asked to create brand new problems that were not solvable by a strong internal model in the loop. All annotators had competitive programming experience. They were instructed to come up with unique problems either from scratch or inspired by programming textbooks whose content was not freely available (e.g., searchable or unlicensed) on the Internet. Annotators could not copy any exercise from a web source or any LLM and only use these sources for inspiration. All sources were cited by annotators and prompts were verified to not match source inspiration. Every prompt is also manually verified as not easily searchable on the web at the time of writing. Each prompt went through additional review to ensure that they were original, hard, and unambiguous. Around a third of the suggested prompts were disqualified for one of these reasons.

Unsolved problems in LBPP
Given a list of unique words each of size k and an n sized word, w, where n is a multiple of k, write a program in Python to determine the number of unique combinations of words in the list that can be concatenated to form an anagram of the word.
Write a function in python that takes as input a recursive function and some parameters. The function should return the number of times the recursive function ran itself when starting it with the provided parameters.
Write a Monte Carlo function in Python to compute the median number of cards you'd need to draw from a deck such that the sum equals or exceeds the value V.

Table 1: Sampled prompts in LBPP unsolved by leaders on existing benchmarks. Prompts shortened for brevity.

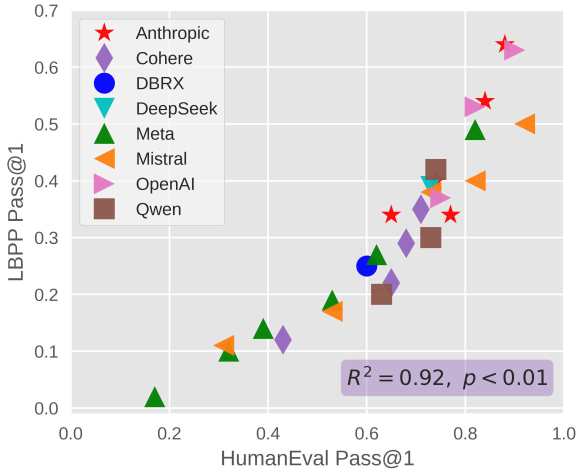
All annotators were paid above minimum wage in their respective countries, and all final prompt-completion pairs were manually reviewed by the authors. This adversarial collection resulted in more difficult problems, with most models solving less than 50% of the dataset.

Initial Results: Table 2 shows Pass@1 on LBPP for a range of models. Notably, leading models for `HumanEval` and `MBPP` perform up to 43% and 27% poorer on LBPP respectively. Additionally, model rankings between either `HumanEval` and `MBPP` update for LBPP, potentially identifying overfitting to public benchmarks when presented with a challenging, unseen test set. LBPP is a similarly reliable indicator of code generation performance than prior benchmarks. In Fig. 3, we observe a strong significant correlation between 'Pass@1' scores of either `HumanEval` or `MBPP` and LBPP. While existing public benchmarks are still a valuable target signal for performance, LBPP is additionally advantageous in that the problems are harder (see Table 2), the dataset is uncontaminated in current training corpora, and prompts bear lower resem-

	Model Name	`HumanEval`	`MBPP`	LBPP	HumanEval→LBPP
Mistral	Mistral 7B	0.31	0.32	0.11	27→26
	Mixtral 8 × 7B	0.53	0.23	0.17	22→23
	Mixtral 8 × 22B	0.73	0.64	0.38	13→11
	Mistral Large	**0.92**	0.74	0.50	1→5
	Codestral 22B	0.82	0.48	0.40	5→9
Meta	Codellama 7B Instruct	0.39	0.37	0.14	25→24
	Codellama 34B Instruct	0.53	0.53	0.19	23→22
	Llama2 7B Chat	0.17	0.19	0.02	28→28
	Llama2 60B Chat	0.32	0.31	0.10	26→27
	Llama3 8B Instruct	0.62	0.44	0.27	20→18
	Llama3 70B Instruct	0.82	0.67	0.49	6→6
OpenAI	GPT-3.5 Turbo	0.75	0.70	0.37	9→12
	GPT-4	0.82	**0.80**	0.53	7→4
	GPT-4o	0.90	**0.80**	0.63	2→2
Antropic	Claude-2	0.65	0.39	0.34	17→15
	Claude-3-Haiku	0.77	0.64	0.34	8→14
	Claude-3-Sonnet	0.74	0.66	0.40	10→8
	Claude-3-Opus	0.84	0.75	0.54	4→3
	Claude-3.5-Sonnet	0.88	0.78	**0.64**	3→1
Qwen	Qwen1.5 72B Chat	0.63	0.53	0.20	19→21
	Qwen1.5 110B Chat	0.73	0.57	0.30	12→16
	Qwen 2 72B Instruct	0.74	0.67	0.42	11→7
Cohere	Command R	0.43	0.46	0.12	24→25
	Command R (Refresh)	0.71	0.55	0.35	15→13
	Command R+	0.65	0.61	0.22	18→20
	Command R+ (Refresh)	0.68	0.61	0.29	16→17
	Deepseek Coder 33B Instr.	0.73	0.66	0.39	14→10
	Databricks DBRX Instr.	0.60	0.57	0.25	21→19

Table 2: Pass@1 results across popular models for zero-shot `HumanEval`, `MBPP` and LBPP. All models perform worse on LBPP than either existing benchmark. Rankings between models **change** between `HumanEval` and LBPP, contrasting to Riddell et al. (2021). Model rankings similarly also change between `MBPP` and LBPP.

(a) `HumanEval`



(b) `MBPP`

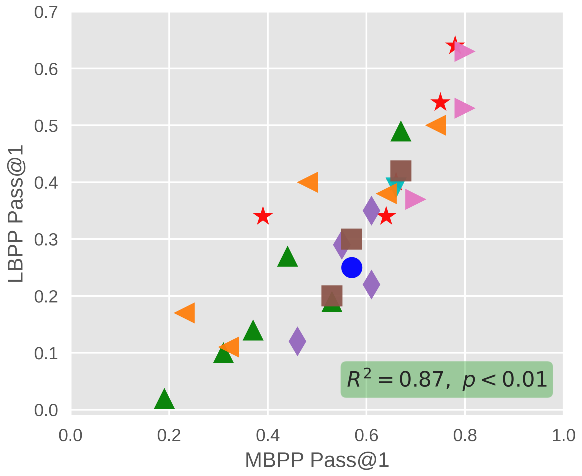


Figure 3: Pass@1 rate of LBPP against (a) `HumanEval` and (b) `MBPP`. LBPP performance correlates with both prior datasets, but is designed to be genuinely unseen by contemporary LLMs.

blance to existing synthetic corpora (see Fig. 2).

Challenges in LBPP: We study common errors and mistakes from multiple models to identify the most challenging features in LBPP. Examples of problems unsolved by all models are shown in Table 1. Considering common errors between Claude 3.5-Sonnet and Command R Refresh, as the best and recently released model respectively, we identify multiple core trends in failure. Of mutual errors: 21% are related to problems on 2D and 3D arrays; 18% are related to graph-oriented algorithms; and 17% are concerning complex programming concepts often used in competition settings. Additional challenging topics include bit arithmetic & manipulation (8%), Pandas data processing (8%), and file IO (8%). The range of shortcomings between all models highlights the variety of domains that future LLMs must master to improve code generation. Overall, the diversity and difficulty of the problems in LBPP challenges even purportedly advanced models with novel and unsolved prompts.

5 Conclusion

We study the cause and effect of data contamination via two popular code generation benchmarks. Our analysis highlights that contamination is likely unavoidable at the LLM scale given the difficulty of filtering every potential permutation of a benchmark dataset. This insight motivates our contribution of LBPP: a novel code generation benchmark to evaluate contemporary LLMs in a contamination-free setting. We are well aware that our decision to release this dataset will make future leakage impossible to control. However, with the context of the fast-paced model development cycles that LLMs are currently undergoing we believe that releasing this increases the trustworthiness and usefulness of this dataset. It is conveniently designed to serve as a drop-in replacement (or addition) of current evaluation sets. On top of its novelty, the more challenging nature of this dataset also provides a cleaner signal for model comparison.

6 Limitations

All the model analysis was done *black-box*, without inspecting the model weights or the training set (except the work on synthetic data). There is no reason why this dataset will not follow the same path as the two studied here. As mentioned in the Conclusion we believe there is more value in that than in an alternative solution (not releasing or keeping it behind an API access).

References

Jacob Austin, Augustus Odena, Maxwell I. Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. 2021. Program synthesis with large language models. *CoRR*, abs/2108.07732.

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*.

Yanai Elazar, Akshita Bhagia, Ian Helgi Magnusson, Abhilasha Ravichander, Dustin Schwenk, Alane Suhr, Evan Pete Walsh, Dirk Groeneveld, Luca Soldaini, Sameer Singh, Hannaneh Hajishirzi, Noah A. Smith, and Jesse Dodge. 2024. What's in my big data? In *The Twelfth International Conference on Learning Representations*.

Leo Gao, Stella Biderman, Sid Black, Laurence Golding, Travis Hoppe, Charles Foster, Jason Phang, Horace He, Anish Thite, Noa Nabeshima, et al. 2020. The pile: An 800gb dataset of diverse text for language modeling. *arXiv preprint arXiv:2101.00027*.

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*.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2024. Livecodebench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974*.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. 2024. SWE-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*.

Denis Kocetkov, Raymond Li, Loubna Ben Allal, Jia Li, Chenghao Mou, Carlos Muñoz Ferrandis, Yacine Jernite, Margaret Mitchell, Sean Hughes, Thomas Wolf, et al. 2022. The stack: 3 tb of permissively licensed source code. *arXiv preprint arXiv:2211.15533*.

Hokyung Lee, Sumanyu Sharma, and Bing Hu. 2024. Bug in the code stack: Can llms find bugs in large python code stacks.

Katherine Lee, Daphne Ippolito, Andrew Nystrom, Chiyuan Zhang, Douglas Eck, Chris Callison-Burch, and Nicholas Carlini. 2022. Deduplicating training data makes language models better. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 8424–8445, Dublin, Ireland. Association for Computational Linguistics.

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*.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. 2022. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097.

Jiawei Liu, Jia Le Tian, Vijay Daita, Yuxiang Wei, Yifeng Ding, Yuhan Katherine Wang, Jun Yang, and Lingming Zhang. 2024a. Repoqa: Evaluating long context code understanding. *arXiv preprint arXiv:2406.06025*.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2024b. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems*, 36.

Meta Llama Team. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*.

Niklas Muennighoff, Qian Liu, Armel Zebaze, Qinkai Zheng, Binyuan Hui, Terry Yue Zhuo, Swayam Singh, Xiangru Tang, Leandro Von Werra, and Shayne Longpre. 2023. Octopack: Instruction tuning code large language models. *arXiv preprint arXiv:2308.07124*.

Allen Riddell, Haining Wang, and Patrick Juola. 2021. A call for clarity in contemporary authorship attribution evaluation. In *Proceedings of the International Conference on Recent Advances in Natural Language Processing (RANLP 2021)*, pages 1174–1179, Held Online. INCOMA Ltd.

Martin Riddell, Ansong Ni, and Arman Cohan. 2024. Quantifying contamination in evaluating code generation capabilities of language models. *arXiv preprint arXiv:2403.04811*.

Cohere Embedding Team. 2024. Cohere embed-english-v3.0.

Yuxiang Wei, Federico Cassano, Yifeng Ding, Naman Jain, Harm de Vries, Leandro von Werra, Arjun Guha, and Lingming Zhang. 2024. Starcoder2-instruct: Fully transparent and permissive self-alignment for code generation.

Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. 2023. Magicoder: Source code is all you need. *arXiv preprint arXiv:2312.02120*.

Can Xu, Qingfeng Sun, Kai Zheng, Xiubo Geng, Pu Zhao, Jiazhan Feng, Chongyang Tao, Qingwei Lin, and Daxin Jiang. 2023. Wizardlm: Empowering large pre-trained language models to follow complex instructions. In *The Twelfth International Conference on Learning Representations*.

Zhaojian Yu, Xin Zhang, Ning Shang, Yangyu Huang, Can Xu, Yishujie Zhao, Wenxiang Hu, and Qiufeng Yin. 2023. Wavecoder: Widespread and versatile enhanced instruction tuning with refined data generation. *arXiv preprint arXiv:2312.14187*.

Hugh Zhang, Jeff Da, Dean Lee, Vaughn Robinson, Catherine Wu, Will Song, Tiffany Zhao, Pranav Raja, Dylan Slack, Qin Lyu, et al. 2024. A careful examination of large language model performance on grade school arithmetic. *arXiv preprint arXiv:2405.00332*.

A Appendix

MBPP test prompt	**`evol-instruct` prompt**
Write a function to find the perimeter of a square	Calculate the perimeter of a square
Write a python function to find the volume of a triangular prism	How can I use JavaScript to calculate the volume of a triangular prism with an irregular base and uneven side lengths?
Write a function to convert snake case string to camel case string.	Convert the sentence to camel case.

Table 3: Similarity in prompts between `MBPP` evaluation dataset and `evol-instruct` synthetic training dataset.

Table 4: Original human evaluation prompts with the completion from a major LLM provider.

Original human-eval prompt

```
def is_happy(s):
"""You are given a string s.
Your task is to check if the string is happy or not.
A string is happy if its length is at least 3
and every 3 consecutive letters are distinct
For example:
is_happy(a) => False
is_happy(aa) => False
is_happy(abcd) => True
is_happy(aabb) => False
is_happy(adb) => True
is_happy(xyy) => False """
```

Submitted Prompt

Continue this code. Do not add any test, just continue it:

```
def is_happy(s):
""" You are given a string s. Your task is to check if the str
```

Opus completion

Here's the continuation of the code without adding any tests:

```
def is_happy(s):
"""
You are given a string s. Your task is to check if the string
is happy. A string is called happy if its length is at least 3
and every 3 consecutive letters are distinct

[...]
```

Original human-eval prompt

```
def count_upper(s):
""" Given a string s, count the number of uppercase vowels
in even indices.

For example:
count_upper('aBCdEf') returns 1
count_upper('abcdefg') returns 0
count_upper('dBBE') returns 0
"""
```

Submitted Prompt

Complete this instruction, that is taken from the Human Eval test set:

```
def count_upper(s):
""" Given a string s, count the number of
```

Opus completion

Here is the completion of the given function signature and docstring to count the number of uppercase letters in a string:

```
def count_upper(s):
"""
Given a string s, count the number of uppercase vowels in
the string.

[...]
```

Original human-eval prompt

```
def any_int(x, y, z):
''' Create a function that takes 3 numbers.
Returns true if one of the numbers is equal to the sum of the
other two, and all numbers are integers. Returns false in any
other cases.

Examples:
any_int(5, 2, 7) -> True
any_int(3, 2, 2) -> False
any_int(3, -2, 1) -> True
any_int(3.6, -2.2, 2) -> False '''
```

Submitted Prompt

continue this code.

```
def any_int(x, y, z):
"""Create a function that takes 3 numbers. Return true i
```

Opus completion

```
def any_int(x, y, z):
"""
Create a function that takes 3 numbers.
Returns true if one of the numbers is equal to the sum of the
other two numbers. Otherwise returns false.
"""
[...]
```

Original human-eval prompt

```
def order_by_points(nums):
""" Write a function which sorts the given list of integers
in ascending order according to the sum of their digits.

Note: if there are several items with similar sum of their
digits, order them based on their index in original list.

For example:

order_by_points([1, 11, -1, -11, -12]) == [-1, -11, 1, -12, 11]
order_by_points([]) == []
"""
```

Submitted Prompt

You will be given an instruction to complete a python function. This instruction is cut in half, so you need to complete it. No need of generating the final code, only the instruction is required.
The incomplete instruction is:

```
def order_by_points(nums):
""" Write a function which sorts the given list of integers in
ascending order according
```

Opus completion

Here's the completed instruction for the Python function:

```
def order_by_points(nums):
""" Write a function which sorts the given list of integers in
ascending order according to the sum of their digits.
If two integers have the same sum of their digits, order them
based on their index in the original list.
"""
```

Table 5: Most similar prompt in `evol-instruct` for a random sample of `HumanEval` prompts.

<table>
<tr><th><code>HumanEval</code> prompt</th><th>Closest <code>evol-instrct</code> prompt</th></tr>
<tr><td>

```
Write a Python function `longest(strings: List[str]) -> Optional[str]` to solve the following
problem:
Out of list of strings, return the longest one. Return the first one in case of multiple
strings of the same length. Return None in case the input list is empty.
>>> longest([])
>>> longest(['a', 'b', 'c'])
'a'
>>> longest(['a', 'bb', 'ccc'])
'ccc'
```

</td><td>

```
Complete the code below, considering an augmented depth of inquiry and maintaining brevity:
from typing import List, Optional


def longest(strings: List[str]) -> Optional[str]:
    """ From a list of strings, return the longest one. For multiple strings with equal
    length, return the first. For an empty list, return None.
    >>> longest([])

    >>> longest(['a', 'b', 'c'])
    'a'
    >>> longest(['a', 'bb', 'ccc'])
    'ccc'
```

</td></tr>
<tr><td>

```
Write a Python function `make_a_pile(n)` to solve the following problem:
Given a positive integer n, you have to make a pile of n levels of stones.
The first level has n stones.
The number of stones in the next level is:
- the next odd number if n is odd.
- the next even number if n is even.
Return the number of stones in each level in a list, where element at index
i represents the number of stones in the level (i+1).
Examples:
>>> make_a_pile(3)
[3, 5, 7]
```

</td><td>

```
Please complete the following code with added difficulty:

def make_a_pile(n, pattern):
    """
    Given a positive integer n, you have to make a pile of n levels of stones.
    The first level has n stones. The number of stones in the next level is determined
    by the given pattern 'odd' or 'even':
        - If pattern is 'odd', add the next odd number to the previous level stones.
        - If pattern is 'even', add the next even number to the previous level stones.
    Return the number of stones in each level in a list, where element at index
    i represents the number of stones in the level (i+1).

    Examples:
    >>> make_a_pile(3, 'odd')
    [3, 5, 7]
    >>> make_a_pile(3, 'even')
    [3, 6, 9]
    """
```

</td></tr>
<tr><td>

```
Write a Python function `x_or_y(n, x, y)` to solve the following problem:
A simple program which should return the value of x if n is
a prime number and should return the value of y otherwise.
Examples:
for x_or_y(7, 34, 12) == 34
for x_or_y(15, 8, 5) == 5
```

</td><td>

```
Complete the subsequent lines of programming code:
/*
An elementary program intended to, given a single n value, decipher between two
distinct possibilities:
- If the provided n is a numerical prime, the program should yield the
output value equivalent to variable x.
- Should the n fail to meet the prime criteria, the program should then
present the value contained within variable y as its output.

Examples:
Upon running x_or_y(7, 34, 12) the output will be 34
While executing x_or_y(15, 8, 5) the output will be 5

*/
#include<stdio.h>
using namespace std;
int x_or_y(int n,int x,int y){
```

</td></tr>
</table>