On Leakage of Code Generation Evaluation Datasets

Anonymous EMNLP submission

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.

Key to our findings is a new dataset of 161 prompts with their associated python solutions, dataset which we plan to release with this paper under a research license.

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 academic and industry labs—use at least one of these two 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. On one side, obtaining competitive numbers comes with significant scientific and economic reward—made increasingly easy with the proliferation of public replicas of these datasets. However, this prevalence has led to data leakage beyond the original evaluation scope, i.e., *data contamination*, and once this evaluation data *contaminates* model training, the validity of the metrics as a measure of generalization capability becomes unreliable. If a model has been trained on the same data we use for out-of-distribution generalization (or is selected based on its performance on that data), we break an implicit tenet of how model capability can 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 these two benchmarks have contaminated most large LLMs, which we define as any procedure that leaked those datasets *during* model training. The most obvious method of contamination is presence inside training data, and we provide evidence that it is highly probably that this occurs at a scale too large to be avoidable. A second possibility is that contamination happens indirectly through the use of synthetic data—a widespread paradigm used in particular to increase code capabilities by generating additional code training tokens. Finally, we argue that final model selection might have been overly influenced by their performance on these datasets, overfitting to performance on these metrics over general-purpose code-oriented skills.

To measure this contamination, we propose **Less Basic Python Problems** (LBPP), a code generation benchmark similar to `HumanEval` and `MBPP` in style and scale, but more difficult. LBPP is similarly portable, but is produced in a manner to reduce any likelihood of leakage into present code training data. We contribute LBPP to act as a genuinely held-out test set to measure *current* code generation capability, and potential overfitting to `HumanEval` and `MBPP`.

2 Related Work

`HumanEval` (Chen et al., 2021) and `MBPP` (Austin et al., 2021) remain the most reported results on public leaderboards, but others 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. Translation into other programming languages exist for those datasets (Muennighoff et al., 2023; Cassano et al., 2022), as well as versions with additional tests (Liu et al., 2024).

(Jain et al., 2024) proposed a continuously updated set of leetcode to improve dataset challenge by including harder and novel (unseen) prompts. (Jimenez et al., 2024) aims for challenging software engineering problems, that require understanding of full repositories. In a similar vein, RepoQA[1] and Bug In The Code Stack[2] focus on understanding long contexts within code tasks. One proposed solution is to use hidden evaluation sets (Zhang et al., 2024), however, these do 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). Different from our analysis however, they conclude: "we do not find the performance rankings of the models to change with decontaminated results".

3 Possible sources of contamination

We provide three hypotheses—with evidence for each one—on why existing models might 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 might be that modern LLMs are just 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 results. For code tasks, it is very expensive to curate datasets of natural language to code instructions (one example generally costing several dozen US dollars). For any group aiming to minimize data cost to improve coding performance, the dollar value of creating new datasets can be very high. This leads to a common practice of web scraping code-oriented resources (e.g., GitHub or Stackoverflow) for data. However, these resources are also likely sources of contamination. Since the release of `HumanEval` and `MBPP` in 2021 these datasets have been branched, re-used and copied all across the Internet. The small data size and portability of such benchmarks encourages replication within code repositories. For example, searching for the prompts from `HumanEval` on GitHub returns a hit in all cases—the median hits is 99 and the minimum 43 (see Fig. 2). In many cases, these hits are exact duplicates and indications of a fork of the original dataset.

While decontamination of training sets is becoming more common, present decontamination filters designed for natural text adapts 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.[3] The same shortcomings of decontamination efforts 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`.

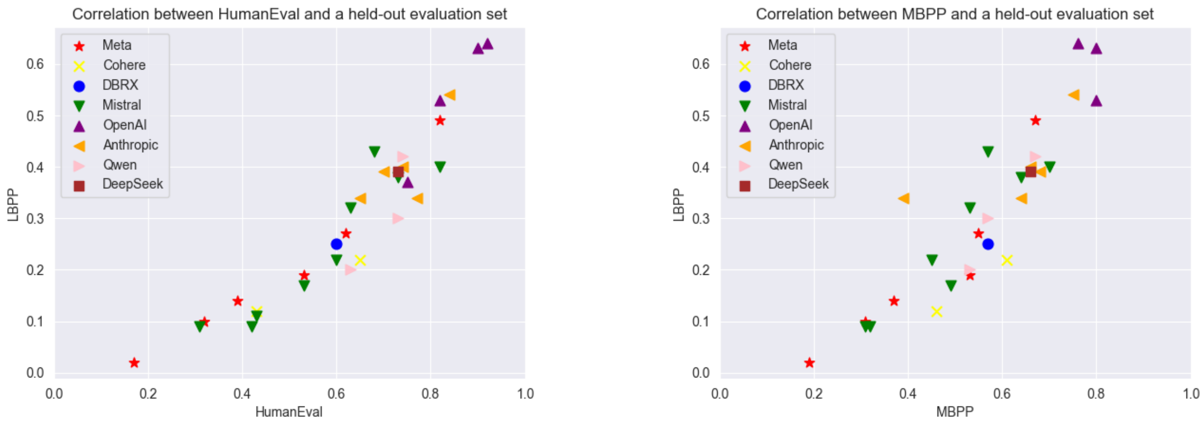
The recent exploration of Riddell et al. (2024) aims to quantify the proportion of this data leakage in existing datasets using plagiarism tools specifically designed for code. Even when static training datasets are cleaned, contamination may persist. Entities who serve models through an API may encounter these benchmark tasks 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 subtle phrasing variations 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 phenomena, we prompted one popular commercial system with partial prompts from `HumanEval` that were designed to keep the instruction under-specified. Table 2 in the Appendix shows the outcome and evidence that—despite the ambiguity of the prompt—the resulting completion matches exactly the gold solution from the test set of `HumanEval`.

---

[1] `https://github.com/evalplus/repoqa`

[2] `https://github.com/HammingHQ/bug-in-the-code-stack`

[3] E.g., compare the instruction "`return true if the string is a float`" with "`return true if the string is a verb`".



(a) `HumanEval` (b) `MBPP`

Figure 1: Pass@1 rate of popular datasets and the 161 prompts in LBPP.

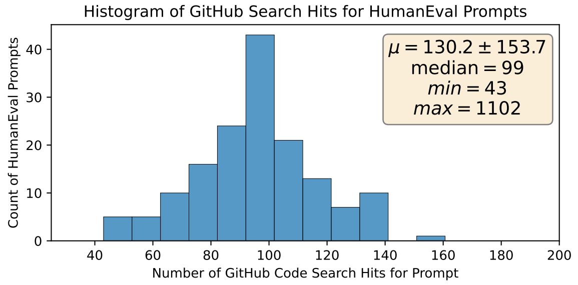


Figure 2: Histogram (excluding outliers) of occurrences for `HumanEval` prompts in public GitHub repositories. Every prompt occurs at least 43 times.

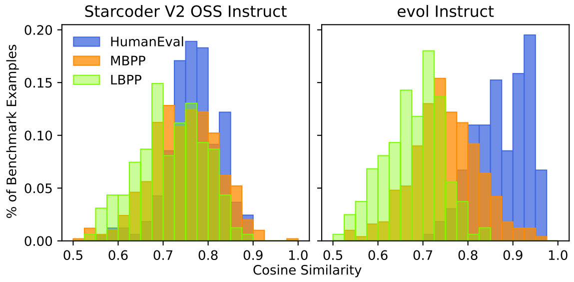


Figure 3: Histogram of cosine similarities for prompts in `HumanEval`, `MBPP` and LBPP relative to two popular synthetic code training datasets. We note the high similarity between most `HumanEval` prompts to *evol-instruct*, and how LBPP has reduced overall similarity to either training dataset.

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).

A typical pipeline generally 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 that pass 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.

`evol-instruct` for example comprises 110k complex query prompts coupled with completions from numerous closed and open-source models.[4] It is widely used by many code LLMs such as WizardCoder (Xu et al., 2023). Prior reports (Yu et al., 2023, page 8), (Wei et al., 2023, page 4) discuss an apparent high similarity between some of those examples in `evol-instruct` and `HumanEval`. We extend this analysis by studying the similarity between embedded representations of the *prompts*[5] of `HumanEval` and `MBPP` with nearest neighbors from *evol-instruct* and *Starcoder-Instruct*. Fig 3 highlights widespread similarity between evaluation data and synthetic training datasets 'Starcoder V2 OSS Instruct' and 'evol instruct'. Even if unintentional, this contamination further damages the utility of these benchmarks as a held-out test of code generation capability.

Strikingly, we saw that training on this dataset can increase performance of one of our models by 14 absolute points on `HumanEval` (from 0.52 pass@1 to 0.66), while only by 1 point on `MBPP` (from 0.52 to 0.53). Inspecting closest neighbours for almost all examples in `HumanEval` (see examples in Table 3 in the Appendix)—we identify a se-

---

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

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

mantically equivalent version in `evol-instruct`.

Even when the original evaluation datasets were not used as inspiration for the creation of the synthetic datasets the simple nature of the evaluation datasets might make duplication unavoidable. There are only so many short natural language prompts describing typical interview-style programming questions that can be used. While synthetically generated prompts may not be explicitly based on provided examples from a given test set in the prompt context, the massive scale of these datasets ($238k$ instances for StarCoder-Instruct prior to deduplication) runs the risk of exhausting the possible number of variations on questions. Table 1 shows examples where generated prompts are extremely similar to questions in the `MBPP` test set. Note that despite that similarity training on this dataset did not improve substantially performance on `MBPP`.

3.3 Overfitting to test sets

The exaggerated importance of these 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, there is weak evidence for these benchmarks correlating with 'solving code generation'. While the meaning and measurement of this unscientific objective is subject to constant revision, selecting for optimal `HumanEval` performance may be akin to p-hacking in other fields. This practice can be justified by assuming that these benchmarks are the new dev sets, while the true test is the usage of users over time. The risk remains however that some models overfit to those test sets more than others, distorting perception on relative performance.

In order to measure this, we created **Less Basic Python Problems (LBPP)**, a dataset of 161 code completion problems in the style of `HumanEval`. Human annotators were instructed to create totally fresh problems, which were not solvable by an internal model[6] they had access to. Annotators had competitive programming experience and could use programming books as inspiration, but were instructed not to copy existing solutions on the Internet and not to use any LLMs. 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. Results on a selection of models are in Table 4 in the Appendix.

Using this dataset we can correlate the performances of existing models on them against performance of the well-known benchmarks. The two plots comparing them to `HumanEval` and `MBPP` can be seen in Fig. 1. There is a clear correlation on both data-sets, indicating that the public benchmarks are still a valuable target signal. However, when zooming in — in particular for models from the same family — the correlation often becomes negative, which might indicate that the selected checkpoint to release performs better on those public datasets while under-performing on new datapoints. Note in particular in Fig. 1a the crowded space between 0.75 and 0.8 of the x-axis (pass@1 of `HumanEval`). Despite very similar performance on that public evaluation set, the numbers of LBPP vary wildly. On the other side, in Fig. 1b a vertical line can be traced at around 0.38 of LBPP that crosses various points – indicating maybe an overfit to `MBPP`.

4 Conclusion

We study the cause and effect of data contamination by 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 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 trustworthiness and usefulness of this dataset. It is conveniently designed to serve as drop-in replacement (or addition) of current evaluation sets. On top of its newness, the more challenging nature of this dataset also allows it to provide more signal for model comparison.

5 Limitations

All the model analysis was done *black-box*, without inspecting the model weights or the training set (excepting the work on synthetic data). There is no

---

[6] We will update references to internal models in future revisions.

reason why this dataset will not follow the same path than 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*.

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*.

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

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*.

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*.

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 1: Examples of similar prompts from the test set of `MBPP` found in evol_instruct (Xu et al., 2023).

Table 2: 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 3: Most similar prompt in `evol-instruct` for a non-handpicked selection of `HumanEval` prompts.

`HumanEval` prompt | **Closest `evol-instrct` prompt**

```
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'
```

```
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'
```

```
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]
```

```
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]
    """
```

```
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
```

```
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){
```

Model name	**Family**	**pass@1**
claude-2	Anthropic	0.65
claude-2.1	Anthropic	0.70
claude-3-haiku	Anthropic	0.77
claude-3-sonnet	Anthropic	0.74
claude-3-opus	Anthropic	0.84
Command R+	Cohere	0.65
Command R	Cohere	0.43
DBRX Instruct	DBRX	0.60
deepseek-coder-33b-instruct	DeepSeek	0.73
codellama-7b-instruct	Meta	0.39
codellama-34b-instruct	Meta	0.53
codellama-70b-instruct	Meta	0.51
llama2-7b-chat	Meta	0.17
llama2-70b-chat	Meta	0.32
llama3-8b-instruct	Meta	0.62
llama3-70b-instruct	Meta	0.82
Mistral-7b-instruct	Mistral	0.31
Mistral-7b-instruct v2	Mistral	0.42
Mistral-7b-instruct v3	Mistral	0.43
Mixtral 8x7B	Mistral	0.53
mistral-small	Mistral	0.63
mistral-medium	Mistral	0.60
mistral-large	Mistral	0.68
Mixtral 8x22B	Mistral	0.73
Codestral 22B	Mistral	0.82
gpt-3.5-turbo 01/25	OpenAI	0.75

Table 4: Results of pass@1 rate on the LBPP dataset for a selection of models.