Quantifying Contamination in Evaluating Code Generation Capabilities of Language Models

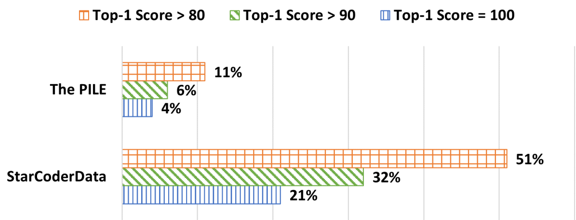
Martin Riddell **Ansong Ni** **Arman Cohan**

Yale University

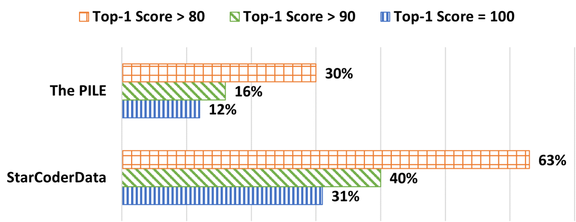
{martin.riddell, ansong.ni, arman.cohan}@yale.edu

Abstract

While large language models have achieved remarkable performance on various code generation benchmarks, there have been growing concerns regarding potential contamination of these benchmarks as they may be leaked into pretraining and finetuning data. While recent work has investigated contamination in natural language generation and understanding tasks, there has been less extensive research into how data contamination impacts the evaluation of code generation, which is critical for understanding the robustness and reliability of LLMs in programming contexts. In this work, we perform a comprehensive study of data contamination of popular code generation benchmarks, and precisely quantify their overlap with pretraining corpus through both surface-level and semantic-level matching. In our experiments, we show that there are substantial overlap between popular code generation benchmarks and open training corpus, and models perform significantly better on the subset of the benchmarks where similar solutions are seen during training. We also conduct extensive analysis on the factors that affects model memorization and generalization, such as model size, problem difficulty, and question length. We release all resulting files from our matching pipeline for future research[1].



(a) Data contamination on the MBPP benchmark.



(b) Data contamination on the HumanEval benchmark.

Figure 1: Quantifying data contamination for the PILE and STARCODERDATA corpus on two popular benchmarks, MBPP and HumanEval. "Top-1 Score" denotes the similarity score between the gold solution and the most similar program found in the training corpus.

1 Introduction

The compute requirements (encompassing both model size and data volume) for training large language models (LLMs) has grown significantly over the years, correlating with consistent observed enhancements in model performance in both language (Kaplan et al., 2020; Hoffmann et al., 2022) and code (Ni et al., 2023) generation tasks. Larger models trained on larger training corpora tend to lead to an increased risk of *data contamination* of the expected output, which we refer to as instances of evaluation benchmark data appearing within the data used during the training of models. LLMs tend to perform better on evaluation samples that resemble the documents and instances encountered during training (Kandpal et al., 2022a; Razeghi et al., 2022; Magar and Schwartz, 2022), and are more likely to emit memorized training data when they have seen it multiple times (Kandpal et al., 2022b; Carlini et al., 2023). Recent papers have also shown evidence that LLMs are possibly contaminated (Golchin and Surdeanu, 2023; Yang et al., 2023), which limits our understanding of their generalization capabilities to unseen inputs.

Despite significant research into data contamination in natural language (NL) benchmarks (Golchin and Surdeanu, 2023; Chang et al., 2023; Blevins and Zettlemoyer, 2022; Dodge et al., 2021; Deng

[1]Code and data available at https://github.com/yale-nlp/code-llm-contamination

Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 14116–14137
August 11-16, 2024

et al., 2023), there's been relatively little exploration into how this issue affects the evaluation of *code generation* capabilities in LLMs. We posit that the fundamental disparities between NL and programs warrant a deeper examination. Recent studies, such as work by Karmakar et al. (2022); Ranaldi et al. (2024), suggest that code-based LLMs may demonstrate patterns of memorization, underscoring the need for scrutiny into their generalization capabilities to unseen cases. Key distinctions between code and NL include the critical role of syntax and the variable requirements for naming functions and variables across different programs. These differences lead us to argue that traditional surface-level comparisons might not be adequate for identifying contaminated data points.

In this paper, we propose a pipeline to measure the overlap between code generation benchmarks and pretraining corpus of code LLMs, incorporating both surface-level and semantic-level code matching. As a result of the exhaustive search amoung the training corpus with our pipeline, we provide a precise quantification of the examples whose solutions are seen during training, for popular code generation benchmarks as MBPP (Austin et al., 2021) and HumanEval (Chen et al., 2021). We study two open pretraining corpus which contain code, the PILE (Gao et al., 2020) and STARCODERDATA (Li et al., 2023), as well as three model series trained on either corpora, StarCoder-Base (Li et al., 2023), Pythia (Biderman et al., 2023) and CodeGen-NL (Nijkamp et al., 2023). Our results show severe contamination of the widely used MBPP and HumanEval benchmarks within the PILE and STARCODERDATA corpora, as shown in Fig. 1, with models performing significantly better on questions that the models have seen the same or similar program solutions to. We perform thorough analysis on factors that may affect model memorization and generalization such as model sizes and difficulty of the questions. We also include a case study on outliers, to provide a more comprehensive understanding of model behavior given different levels of exposure to test data.

2 Methodology

To quantify data contamination for code LLMs, we first introduce methods used to measure program similarity from surface- and semantic-level in § 2.1. Next, in § 2.2 we describe how to combine similarity measurements to identify the overlapping programs in the training data and test benchmarks as well as introduce how to quantify data contamination based on the similarity scores and the number of appearances of similar programs seen during the course of training.

2.1 Measuring Program Similarity

While most popular code generation benchmarks focus on generating functions, the training data are often chunked by files, which may contain multiple functions or classes. This means that document-level deduplication techniques (e.g., Allamanis, 2019) cannot be used effectively, as other programs within the document may add too much noise. Thus we opt to perform substring-level matching, which is much more computationally heavy but also more accurate than methods used by previous work (Lee et al., 2022; Peng et al., 2023; Kandpal et al., 2022a). More specifically, we use a sliding window to scan the training data character-by-character and compute its similarity scores with gold solutions in the benchmarks. To maximize the recall of possible contaminated examples in coding benchmarks, we employ both surface- and semantic-level similarity measurements.

Surface-Level Similarity. To measure surface-level similarity between programs, we use the *Levenshtein similarity score* (Sarkar et al., 2016), which is the Levenshtein edit distance (Levenshtein, 1965) normalized by the length of both the source and target strings. We selected the Levenshtein similarity score as the first step in our pipeline because it is an easy-to-compute and intuitive measurement that can handle surface-level fuzzy matches between two programs. While the Levenshtein edit distance has been used before to deduplicate datasets at a file level (Chowdhery et al., 2022), we perform it on a substring level. An example of this can be found in Fig. 2.[2]

Semantic Similarity. While the surface-level similarity metrics can easily capture similar programs in surface form, two semantically similar or even identical programs can have very different surface form due to different identifiers (*e.g.,* variable names) or whitespace characters. Therefore, finding semantically similar programs is also crucial for measuring contamination and understanding the generalization capabilities of the mod-

[2]we use the `rapidfuzz` python library to calculate the similarity score https://pypi.org/project/rapidfuzz/.

```
Write a python function to find the
    minimum number of squares whose sum
    is equal to a given number.
```

(a) Problem Description.

```
def get_Min_Squares(n):
    if n <= 3:
        return n;
    res = n
    for x in range(1,n + 1):
        temp = x * x;
        if temp > n:
            break
        else:
            res = min(res,1 +
    get_Min_Squares(n  - temp))
    return res;
```

(b) Gold Program

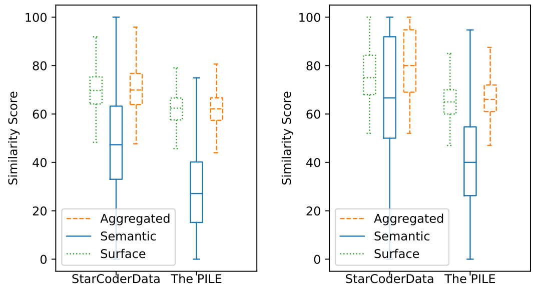
```
def getMinSquares(n):
#     if n <= 3:
#         return n
#     res = n
#     for x in range(1, n+1):
#         temp = x * x
#         if temp > n:
#             break
#         else:
#             res = min(res, 1 +
    getMinSquares(n - temp))
```

(c) Matched program in STARCODERDATA. Surface-level similarity = 91; semantic-level similarity = 0.

Figure 2: Example where surface-level matching works better than semantic-level. Because most of the program is commented out, the semantic-level similarity score is 0 despite the programs being otherwise identical.

els. To measure semantic similarity between programs, we adopt the *Dolos toolkit* (Maertens et al., 2022), which is a source code plagiarism detection tool for education purposes. Dolos first uses `tree-sitter`[3] to tokenize and canonicalize the program into representations of abstract syntax trees (ASTs), then computes a similarity score representing the semantic-level similarity based on the k-gram matching between source and target programs. Since Dolos measures similarities based on the ASTs, non-semantic changes that greatly decrease the Levenshtein similarity scores, such as indentations and variable/function names, will not affect the scores calculated by Dolos. An example of this can be found in Fig. 4. Dolos was also used in previous works for detecting intellectual property violations (Yu et al., 2023).

[3]https://tree-sitter.github.io/tree-sitter/



(a) Top-10 Scores. (b) Top-1 Score.

Figure 3: Distribution of different similarity scoring methods on the MBPP dataset. Similar results for HumanEval are shown in Fig. 11.

2.2 Quantifying Data Contamination

For each problem and its gold solution in the test benchmark (*e.g.,* MBPP), we would like to determine the most similar programs that the models have seen during training. However, this would require us to perform a pair-wise comparison with all programs in the training data using the similarity score metrics mentioned in §2.1. Because training data is usually on the scale of hundreds of gigabytes to terabytes, it is computationally expensive[4] to run surface-level matching methods; running the code-specific semantic matching methods on the entire training dataset is even more computationally prohibitive.

Aggregating Similarity Scores. We use a two-stage process to analyze test examples and their correct (gold standard) program solutions. First, we measure the surface-level similarity by calculating Levenshtein scores. This involves comparing all substrings of the same length as the gold solution across all relevant files in specific subsets of our dataset (see § 3.1 for details). We keep the top 500 programs[5] with the highest Levenshtein similarity scores for each test example for the next step. The similarity scores found by searching the PILE and STARCODERDATA for gold programs from the MBPP benchmark are shown in Fig. 3, along with a comparison between results found using only the top score and those found using an average of the top 10 scores.

[4]We estimate that it will take $5.2*10^5$ CPU hours to search just the Python files from STARCODERDATA for MBPP.

[5]This is determined by a combination of automatic and manual inspection. For example, at the 500^{th} most similar program from STARCODERDATA for MBPP, 95% of them have a similarity score < 72, which is no longer relevant by human inspection.

```
Write a python function to determine
    whether all the numbers are
    different from each other or not.
```

(a) Problem description.

```python
def test_distinct(data):
  if len(data) == len(set(data)):
    return True
  else:
    return False;
```

(b) Gold Program.

```python
def is_simple(graph):
  if len(graph) == len(set(graph)):
    return 1.0
  else:
    return 0.0
```

(c) Matched program in STARCODERDATA. Surface-level similarity = 79; semantic-level similarity = 93.

Figure 4: Example where semantic-level similarity works better than surface-level. The ASTs of two programs are identify despite different variable names.

With the top 500 programs with the highest Levenshtein similarity scores from the training data, we further compute the semantic similarity scores with the gold programs using Dolos. Then the *aggregated similarity score* is computed as the maximum of the surface-level similarity score ($S_{\texttt{surface}}$) and semantic similarity score ($S_{\texttt{semantic}}$) similarity scores:

$$S(p, p^*) = \max(S_{\texttt{surface}}(p, p^*), S_{\texttt{semantic}}(p, p^*))$$

This aggregated similarity score is a simple and intuitive way to reflect how programs can be similar both from their surface form and semantics.

3 Experimental Setup

We select two of the most popular public pretraining corpora for general LLM and code LLMs, namely the PILE (Gao et al., 2020) and STARCODERDATA (Li et al., 2023), and three series of popular open-source models, *i.e.,* Pythia (Biderman et al., 2023), CodeGen-NL (Nijkamp et al., 2023) and StarCoderBase[6](Li et al., 2023). For the coding benchmark, we opt to study MBPP (Austin et al., 2021) and HumanEval (Chen et al., 2021) due to their popularity. We introduce them in more detail in the following subsection.

[6]StarcoderBase refers to the models originally trained on STARCODERDATA (Li et al., 2023). These models were further finetuned on Python code to create the Starcoder models.

3.1 Models and Pretraining Data

We select the models by the following criteria: *1)* The pretraining data for the models must be publicly available; *2)* To ensure non-trivial performance on the coding benchmarks, such models must have Python code in their pretraining data; *3)* Additionally, we do not consider any models that are instruction-tuned, or trained with reinforcement learning from human feedback (*i.e.,* RLHF), as it is hard to quantify the effect of such instruction-tuning/human preference data along with the pretraining corpus. Based on these criteria, we study the following three model series in this work:

The PILE and Pythia. Pythia (Biderman et al., 2023) is a suite of 16 LLMs intended to facilitate research in many areas. All models are trained on the PILE dataset (Gao et al., 2020), with their size ranging from 70M to 12B parameters. We used the 1.4B, 2.8B, 6.9B, and 12B models for this study. We use the GitHub split of the training dataset, which has a raw size of 95.16 GiB.

The PILE and CodeGen-NL. Another series of models that are trained with PILE is CodeGen-NL (Nijkamp et al., 2023), and we study the 350M, 2B, 6B, and 16B versions of it. Though stronger CodeGen models are available via further training on more code data, the exact copy of such data is not publicly released thus we choose to study the CodeGen-NL series. Due to the overlap of training data, we use the results of searching through the GitHub split that we did for the Pythia models.

STARCODERDATA and StarCoderBase. We use the 1B, 3B, 7B and 15.5B StarCoderBase models (Li et al., 2023) that were trained on the STARCODERDATA dataset (Li et al., 2023). Due to the size of the training data, we only search through 60.40 GB within the Python split of its training dataset. The STARCODERDATA dataset is a subset of the STACK (Kocetkov et al., 2022), created by filtering the STACK and applying additional decontamination. The STACK was created from permissively-licensed source code files, and was open-sourced to make the training of code LLMs more reproducible.[7]

3.2 Benchmarks

We measure the data contamination issues for the following two popular coding benchmarks:

[7]It is worth noticing that STACK went through a string-matching-based decontamination process for MBPP and HumanEval, but we are still able to find traces of contamination for these two datasets.

Benchmark	Models	Acc_o	Top-1 Score=100		Top-1 Score>90		Top-1 Score>80	
			% Rm	Acc_d	% Rm	Acc_d	% Rm	Acc_d
MBPP	StarCoderBase-15.5B	41.6	20.8	33.8 (-18.8%)	32.2	32.5 (-22.6%)	50.8	29.7 (-28.6%)
	Pythia-12B	17.8	3.6	17.0 (-4.5%)	6.9	16.6 (-6.7%)	11.4	15.8 (-11.2%)
	CodeGen-NL-16B	19.6	3.6	18.4 (-6.1%)	6.9	17.4 (-11.2%)	11.4	16.5 (-15.8%)
HumanEval	StarCoderBase-15.5B	30.5	18.9	22.6 (-25.9%)	39.6	15.2 (-50.2%)	63.4	20.0 (-34.4%)
	Pythia-12B	9.8	12.2	4.2 (-57.1%)	15.9	2.9 (-70.4%)	29.9	1.7 (-82.7%)
	CodeGen-NL-16B	14.6	12.2	8.3 (-43.2%)	15.9	5.8 (-60.3%)	29.9	3.5 (-76.0%)

Table 1: Measuring the de-contaminated accuracy (Acc_d) by removing potentially contaminated subsets of MBPP and HumanEval *w.r.t.* different thresholds. "Acc_o" denotes original model accuracy and "% Rm" denotes the percentage of the dataset removed. The *relative* accuracy degradation after de-contamination is shown in brackets.

Models	MBPP			HumanEval		
	$\Uparrow^{10\%}$	$\Downarrow_{10\%}$	$\Delta_{\Updownarrow}$	$\Uparrow^{10\%}$	$\Downarrow_{10\%}$	$\Delta_{\Updownarrow}$
StarCoderBase	72.0	22.0	50.0	75.0	31.3	43.7
Pythia	40.0	8.0	42.0	56.3	0.0	56.3
CodeGen-NL	48.0	6.0	42.0	62.5	0.0	62.5

Table 2: We show the performance gap ($\Delta_{\Updownarrow}$) between the top 10% ($\Uparrow^{10\%}$) and bottom 10% ($\Downarrow_{10\%}$) of programs based on the average of the top-10 aggregated similarity scores. Only the *largest* models are shown for each model series, full results available in Tab. 4.

MBPP (Austin et al., 2021) is a benchmark containing 974 short, crowd-sourced Python programming problems. We use the 500 questions within its test split.

HumanEval (Chen et al., 2021) is a benchmark consisting of 164 hand-written problems. Each problem contains a gold solution.

Notably, these two benchmarks come with gold program solutions, which we use to search the pretraining data as a query. To obtain the model performance and predictions on each of the dataset examples, we use the evaluation framework and model outputs from `L2CEval` (Ni et al., 2023).

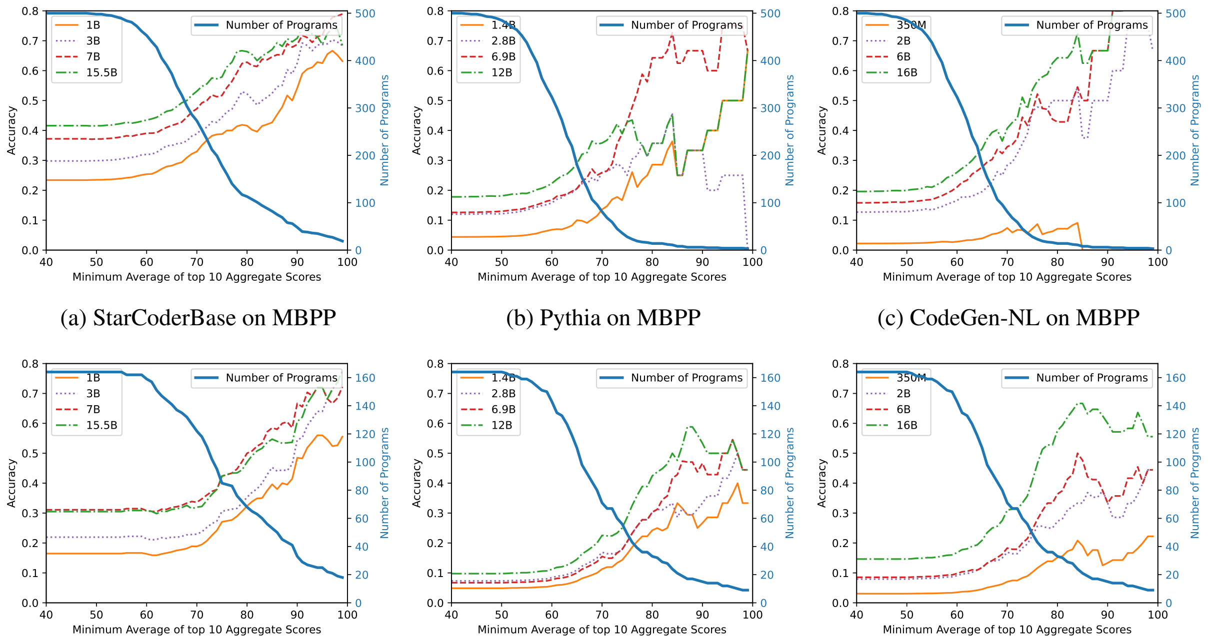
4 Results

In this section, we first present our main results in §4.1, then with several analysis on how the length, difficulty and model sizes affects the our findings in §4.2, and finally we present a case study in §4.3.

4.1 Main Results

3.6% to 20.8% of the solutions are likely seen during training. For an example in the test data (*i.e.,* those of MBPP or HumanEval), we note it as "*seen*" if the aggregated similarity score is 100, *i.e.,* a perfect match exists on the surface- or semantic-level. Results in Fig. 1 show that 12.2% of the solutions in HumanEval have been seen by models trained on the PILE and 18.9% have been seen by models trained on STARCODERDATA. For MBPP, 3.6% of it can be found in the PILE while as much as 20.8% have been seen by models trained on STARCODERDATA. Much less overlap is found for the PILE, as 3.6% of MBPP, but 20.8% of the solutions on MBPP problems have been seen for models trained on STARCODERDATA. These results suggest that a non-trivial part of both MBPP and HumanEval have been seen for the models trained on either the PILE or STARCODERDATA, suggesting a high contamination rate.

Models perform significantly better when similar solutions are seen during training. To observe the effect that having seen a solution during training has on a model, we conduct three different sets of experiments: *1)* We first removed potentially contaminated questions from the dataset, and evaluated the models performance on the new dataset, as seen in Tab. 1. *2)* We also highlight the difference in performance that models have between questions which they have seen similar solutions and questions which they have not. We use the performance gap Razeghi et al. (2022) between the top 10% and bottom 10% of programs based on aggregated similarity scores to do this. The performance gap of the largest models from the chosen model series is shown in Tab. 2, where it can be observed that all three models perform significantly better on questions in the top 10% of compared to questions in the bottom 10%. StarCoderBase-15.5B, which achieves an accuracy of 72% on the top 10% of questions and an accuracy of 22% on the bottom 10% of questions of the MBPP benchmark. The range of similarity scores for each model and benchmark can be found in Fig. 3. *3)* Lastly we discuss the effect of models having seen the solution in §4.2, where we provide an analysis on decoupling memorization and question difficulty.



(a) StarCoderBase on MBPP (b) Pythia on MBPP (c) CodeGen-NL on MBPP

(d) StarCoderBase on HumanEval (e) Pythia on HumanEval (f) CodeGen-NL on HumanEval

Figure 5: Accuracy of different model series evaluated on a subset of examples with increasing overlap with the model's pretraining data. Each subset was obtained by using the x-axis as a threshold for the minimum score obtained by taking the average of the top-10 aggregated similarity scores. We note that as the number of examples decreases, it becomes more likely for the lines to overlap, as can be seen in Fig. 5b.

De-contaminated results. In an attempt to show the impact of seen questions on model performance, we remove potentially contaminated questions from each benchmark, showing the results in Tab. 1. We observe that removing not only questions that have been seen, but also questions where programs similar to the gold program have been seen during training has an adverse effect on model performance. Moreover, from the de-contaminated results, the performance gap between different models could be much smaller. For example, the original accuracy (Acc_o) gap between StarCoderBase-15.5B and Pythia-12B is 23.8%, and after de-contamination, the performance gap is decreased to 13.9%. This indicates that a large part of the performance gap between different models may due to data contamination. While we do not find the performance rankings of the models to change with de-contaminated results, a study with more models might be needed for deriving any general conclusions.

4.2 Analysis

Ablations on model size. We show how model size affects accuracy in regard to the aggregate score in Fig. 5. We observe that larger models tend to perform better than smaller models in their family, indicating that they are not only better at generalization, but also at memorization. We believe that the Pythia and CodeGen-NL models show similar trends in these graphs due to being trained on the same training data. We also note that the noise in each graph grows as there are fewer programs being evaluated on, explaining the 0% accuracy that some models show when evaluated on only questions they have seen 10 or more times during training.

Decoupling memorization and difficulty. We attempt to show that the model performance on seen questions is not just outlying questions that are easier than the remaining questions in the benchmarks. To do so, we compare the overall performance of models on the MBPP and HumanEval benchmarks against their performance on different subsets of questions based on seen and unseen questions in Tab. 3a. We show that while StarCoderBase models perform better on the subset of questions in MBPP that they have seen than on the unseen questions, Pythia and CodeGen-NL models generally perform worse on the same subset of questions. We also provide the sizes of each subset in Tab. 3b, and note the significant overlap between questions in HumanEval that have been seen by

Models	MBPP				HumanEval			
	$\mathcal{D}_{\mathcal{S}}$	$\mathcal{D}_{\mathcal{S}^-}$	$\mathcal{D}_{\mathcal{P}}$	$\mathcal{D}_{\mathcal{P}^-}$	$\mathcal{D}_{\mathcal{S}}$	$\mathcal{D}_{\mathcal{S}^-}$	$\mathcal{D}_{\mathcal{P}}$	$\mathcal{D}_{\mathcal{P}^-}$
StarCoderBase-7B	**63.5**	30.3	33.3	**37.3**	**64.5**	23.3	**75.0**	25.0
StarCoderBase-15.5B	**71.2**	33.8	**55.6**	41.1	**64.5**	22.6	**80.0**	23.6
CodeGen-NL-6B	11.5	**16.9**	**38.9**	14.9	**29.0**	3.8	**45.0**	3.5
CodeGen-NL-16B	11.5	**21.7**	**50.0**	18.5	**48.4**	6.8	**60.0**	8.3

(a) Model performance on different subsets of MBPP and HumanEval.

	$\lvert\mathcal{D}\rvert$	$\lvert\mathcal{D}_{\mathcal{S}}\rvert$	$\lvert\mathcal{D}_{\mathcal{P}}\rvert$	$\lvert\mathcal{D}_{\mathcal{S}\cap\mathcal{P}}\rvert$
MBPP	500	104	18	2
HumanEval	164	31	20	16

(b) The number of questions seen by models trained on STARCODERDATA ($\mathcal{D}_{\mathcal{S}}$) or the PILE ($\mathcal{D}_{\mathcal{P}}$), and the number of programs in both subsets ($\lvert\mathcal{D}_{\mathcal{S}\cap\mathcal{P}}\rvert$).

Table 3: Decoupling memorization and difficulty. $\mathcal{D}_{\mathcal{S}}$ denotes the subset that overlaps with STARCODERDATA, and $\mathcal{D}_{\mathcal{S}^-}$ denotes the complement set (*i.e.*, $\mathcal{D}_{\mathcal{S}^-} = \mathcal{D} - \mathcal{D}_{\mathcal{S}}$). We define $\mathcal{D}_{\mathcal{P}}$ and $\mathcal{D}_{\mathcal{P}^-}$ similarly for the PILE. The better performance amount the two disjoint subsets (*i.e.*, $\mathcal{D}_*$ and $\mathcal{D}_{*^-}$) are in **bold**.

STARCODERDATA or the PILE as compared to the MBPP benchmark. The improved performance of models in the StarCoderBase family on familiar questions in the MBPP benchmark does *not* appear to result from these questions being easier than those they haven't encountered during training. For example, CodeGen-NL-16B has an overall accuracy of 19.6% on the MBPP benchmark, but has an accuracy of only 11.5% on the 104 questions that StarCoderBase has seen. This indicates that models having seen a solution to a question during training significantly increases the performance of models on these questions.

Effect of program length on similarity scoring. One possible concern is that the size of the gold programs could affect the similarity score. Longer strings can have more differences between one another without affecting their aggregated similarity score as much as in shorter strings. To analyze this, we plot the length of every gold program within the MBPP benchmark against the aggregated similarity score of the most similar string within the training dataset used for the StarCoderBase model family in Fig. 6. There does not appear to be a correlation between the length and the aggregated similarity score, or length and accuracy.

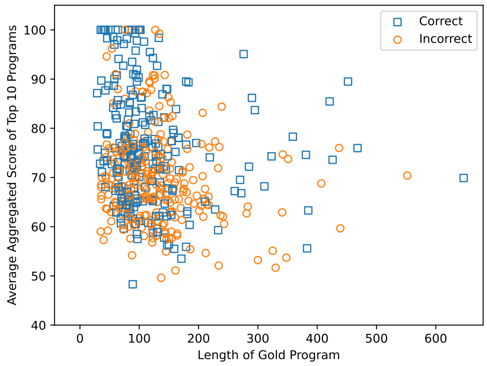


Figure 6: Gold solution length *vs.* overlap with training data *vs.* model prediction correctness, for StarCoderBase-15.5B on MBPP. Similar results for HumanEval are shown in Fig. 12.

4.3 Case Study

Here we present a case study, by showing examples where the models have seen the gold solutions to 10 or more times but still fails to produce a correct solution at test time. Two representative examples are shown as Fig. 7 and Fig. 8. For the first example (Fig. 7), although programs that are similar to the gold program appears multiple times in the pretraining data, understanding the problem description is arguably harder part of the problem. As for the second example (Fig. 8), the gold program is quite simple thus it is not surprising that multiple matches in the training corpus are found, but it may also make it difficult for the model to associate such program with any specific natural language description.

```
Write a function to calculate the sum of
    the positive integers of n+(n-2)+(n
    -4)... (until n-x =< 0).
```

(a) Problem Description

```
def sum_series(n):
  if n < 1:
    return 0
  else:
    return n + sum_series(n - 2)
```

(b) Gold Program.

```
def sum_series(n):
  sum = 0
  for i in range(n):
    sum += i
  return sum
```

(c) Program generated by StarCoderBase.

Figure 7: Example where despite similar solutions appearing 10 or more times in the training corpus, StarCoderBase still fails at test time.

```
Write a function to find the closest
    smaller number than n.
```

(a) Problem Description

```python
def closest_num(N):
  return (N - 1)
```

(b) Gold Program.

```python
def closest_num(n):
  if n % 10 == 0:
    return n - 1
  else:
    return n - (n % 10)
```

(c) Program generated by StarCoderBase.

Figure 8: Another example where despite similar solutions appearing 10 or more times in the training corpus, StarCoderBase still fails at test time.

5 Related Work

Measuring contamination. Our study on the effect of contaminated test questions on accuracy are similar to the work done by Carlini et al. (2020); Henderson et al. (2017); Jiang et al. (2024); Thakkar et al. (2020); Thomas et al. (2020), but instead of perturbing the training dataset, we search through the training datasets for the the gold solutions to the benchmarks. Another line of work in studying memorization is to find documents related to the *output* within a training dataset (Lee et al., 2022; Peng et al., 2023; Kandpal et al., 2022a; Magar and Schwartz, 2022). These works search the training dataset for documents relevant to a string and report the number of relevant documents. While we directly the effects of training data in model outputs, other approaches exist in using the model's weights to find the parts of the training dataset that influenced the model (Han and Tsvetkov, 2022; Grosse et al., 2023). While this paper focuses on searching for contamination in open source models, many models are released without disclosing their training data. To search for contamination in these models, recent papers (Shi et al., 2023; Oren and Meister, 2023; Ranaldi et al., 2024; Deng et al., 2023) use the probabilities of model outputs to observe contamination. This style of approach seems to work primarily when there are multiple copies within the training dataset, and is unreliable at detecting duplication rates of 1 (Oren and Meister, 2023). More recently, Dong et al. (2024) identifies contamination by measuring the peakedness of each model's output distribution via sampling, which works for black-box LLMs but provides less certainty compared with our method.

Plagiarism detection. Plagiarism detection is related to finding similar documents, and some work has already been done on evaluating the similarity of generated programs. Yu et al. (2023) uses two methods, JPlag (Prechelt and Malpohl, 2003) and Dolos (Maertens et al., 2022) to calculate similarity scores between programs. Using the maximum score from the two methods, they determined any two programs with a similarity score greater than or equal to 0.5 to be potential plagiarism. Here, we only use Dolos, due to JPlag's restrictive license.

6 Discussions

Measuring contamination on the outputs. For a generation task, both the input and output are a sequence of tokens, allowing for the contamination of either the input sequence, the output sequence, or their coexistence within the training data to be measured. Given that language models only use the input as context and do not attempt to generate it, we believe it is unlikely that seeing the input during training would help them generate better code at test time. Instead we choose to measure data contamination of the outputs only. We believe that this is a reasonable and arguable the most effective strategy for code generation for the following reasons: *1)* It is common to include the function name and signature in the task description for code generation tasks. After the model copies the function signature, having seen the function during training, it is easy for the model to reproduce the function body at test time. *2)* Since programs are formal languages with strict grammar rules, it is easier to measure the semantic similarity between programs with different surface forms. In our work, we measure the semantic similarity between programs based on their abstract syntax trees. Such semantic comparison would be much harder and more prone to false-negatives for natural language (*i.e.,* the input), as illustrated by recent work (Yang et al., 2023). *3)* We find during our case study that while the models may struggle to reproduce gold programs that they have seen during training due to a misunderstanding of the natural language Fig. 7, there were only a few instances of this found. *4)* While it is indeed the case that the model still needs to associate the natural language description with the code it has seen during training, our results (Tab. 1 and Fig. 1) suggest that there is a strong correlation between output side contamination and model performance, suggesting that having seen the similar code outputs at test time does provide an unfair advantage to the models.

Suggestions for future work. The issue of contamination poses a significant challenge for the future evaluation of large language models' capabilities. Our research suggests that acknowledging the potential for contamination is crucial, especially when utilizing datasets known to be affected, such as HumanEval and MBPP. When creating new datasets, while it is advised to follow our pipeline to decontaminate against the popular pretraining corpus (*e.g.,* GitHub), our results also suggest that it is possible to decouple complexity and memorization by cross referencing the results from models pretrained with different corpus (*i.e.,* results in Tab. 3). Beyond mere surface-level matching, we advocate for a more nuanced approach to decontamination that incorporates understanding the semantics of code when creating pretraining data and new LLMs. Furthermore, model developers should ideally include analyses on the effect of contamination on their evaluations. Given the increasing scale and complexity of state-of-the-art models and dataset used to train these models, developing evaluation benchmarks that are both relevant and completely unseen by models during training seems critical, yet increasingly challenging.

7 Limitations

False negative and positives. Due to the flexibility of programs, there can be multiple correct ways to solve a problem using Python. What we are searching for is only one possible solution presented as the gold solution, and as such we present our findings on the minimum number of questions to which models trained on the PILE and STARCODERDATA have been exposed to. Another source of false negatives (examples we falsely believe to be uncontaminated) is from the search itself. To reduce compute costs, we had to limit our search to relevant splits of the training corpus, which is the GitHub split for the PILE and the Python split for STARCODERDATA. It is possible for the models to see more of similar solutions in other parts of the pretraining corpus (*e.g.,* similar programs in Java for STARCODERDATA). On the other hand, while performing semantic-level comparison helps with the general recall of similar programs, it is possible for it to flag a program as being similar to the gold program despite being quite different, creating false positive example. We present the results for all programs found to be perfect matches to the gold program in § A.4, and some of examples show that false positives do exist.

Different training stages. As mentioned in § 3.1, while we only study the "base" models and their pretraining data in this work, current models are typically trained in multiple stages, including supervised-finetuning, instruction-tuning and RLHF. However, the same methodology should be applicable to training data from these stages as well. At the same time, the size of such data is often several magnitudes smaller than the pretraining data, thus less likely to contain sources of contamination on the example-level. For example, an inspection on the instruction-tuning data for Octocoder (Muennighoff et al., 2023) shows that none of the MBPP examples has a similarity score above 80.

Scarcity of open data. We only include two benchmarks and three models for this study. This is due to the scarcity of commonly used benchmarks that provide a gold program for every question, and models that have an open source training dataset. We hope more open models with open data will become available in the future to fuel further research in this domain.

Inability to retrain model. Ideally, to observe the effect that having seen the answers to questions have on models, we would remove the answers from the training dataset, retrain the model, and compare with the original model. This approach is unfortunately prohibitively expensive given our compute constraints.

8 Conclusion

In this work, we quantify the data contamination issues for two popular code generation benchmarks, namely MBPP and HumanEval. We use both surface-level and semantic similarity to exhaustively search among the pretraining data using gold solutions for these benchmarks. By studying three series of models trained on two different corpus, the PILE and STARCODERDATA, we find that significant portions of the benchmarks have solutions leaked into the pretraining data. Further analysis shows that models perform much better when similar solutions are seen during training, and such correlation is independent of the difficult and length of the problems.

Acknowledgements

The authors would like to thank Arjun Guha, Rui Shen, Yilun Zhao and the late Dragomir Radev for their help, suggestions, and comments for this project. We would also like to thank Hailey Schoelkopf for helping us with the PILE dataset as well as helpful discussions and feedback.

References

Miltiadis Allamanis. 2019. The adverse effects of code duplication in machine learning models of code. In Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward! 2019, page 143–153, New York, NY, USA. Association for Computing Machinery.

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

Stella Biderman, Hailey Schoelkopf, Quentin Gregory Anthony, Herbie Bradley, Kyle O'Brien, Eric Hallahan, Mohammad Aflah Khan, Shivanshu Purohit, Usvsn Sai Prashanth, Edward Raff, Aviya Skowron, Lintang Sutawika, and Oskar Van Der Wal. 2023. Pythia: A suite for analyzing large language models across training and scaling. In Proceedings of the 40th International Conference on Machine Learning, volume 202 of Proceedings of Machine Learning Research, pages 2397–2430. PMLR.

Terra Blevins and Luke Zettlemoyer. 2022. Language contamination helps explains the cross-lingual capabilities of English pretrained models. In Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing, pages 3563–3574, Abu Dhabi, United Arab Emirates. Association for Computational Linguistics.

Nicholas Carlini, Daphne Ippolito, Matthew Jagielski, Katherine Lee, Florian Tramer, and Chiyuan Zhang. 2023. Quantifying memorization across neural language models.

Nicholas Carlini, Florian Tramèr, Eric Wallace, Matthew Jagielski, Ariel Herbert-Voss, Katherine Lee, Adam Roberts, Tom B. Brown, Dawn Xiaodong Song, Úlfar Erlingsson, Alina Oprea, and Colin Raffel. 2020. Extracting training data from large language models. In USENIX Security Symposium.

Kent K. Chang, Mackenzie Cramer, Sandeep Soni, and David Bamman. 2023. Speak, memory: An archaeology of books known to chatgpt/gpt-4.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating large language models trained on code.

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier Garcia, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Diaz, Orhan Firat, Michele Catasta, Jason Wei, Kathy Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. 2022. Palm: Scaling language modeling with pathways.

Chunyuan Deng, Yilun Zhao, Xiangru Tang, Mark Gerstein, and Arman Cohan. 2023. Investigating data contamination in modern benchmarks for large language models.

Jesse Dodge, Maarten Sap, Ana Marasović, William Agnew, Gabriel Ilharco, Dirk Groeneveld, Margaret Mitchell, and Matt Gardner. 2021. Documenting large webtext corpora: A case study on the colossal clean crawled corpus. In Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, pages 1286–1305, Online and Punta Cana, Dominican Republic. Association for Computational Linguistics.

Yihong Dong, Xue Jiang, Huanyu Liu, Zhi Jin, and Ge Li. 2024. Generalization or memorization: Data contamination and trustworthy evaluation for large language models. arXiv preprint arXiv:2402.15938.

Leo Gao, Stella Biderman, Sid Black, Laurence Golding, Travis Hoppe, Charles Foster, Jason Phang, Horace He, Anish Thite, Noa Nabeshima, Shawn

Presser, and Connor Leahy. 2020. The pile: An 800gb dataset of diverse text for language modeling.

Shahriar Golchin and Mihai Surdeanu. 2023. Time travel in llms: Tracing data contamination in large language models.

Roger Baker Grosse, Juhan Bae, Cem Anil, Nelson Elhage, Alex Tamkin, Amirhossein Tajdini, Benoit Steiner, Dustin Li, Esin Durmus, Ethan Perez, Evan Hubinger, Kamil.e Lukovsiut.e, Karina Nguyen, Nicholas Joseph, Sam McCandlish, Jared Kaplan, and Sam Bowman. 2023. Studying large language model generalization with influence functions. ArXiv, abs/2308.03296.

Xiaochuang Han and Yulia Tsvetkov. 2022. Orca: Interpreting prompted language models via locating supporting data evidence in the ocean of pretraining data. ArXiv, abs/2205.12600.

Peter Henderson, Koustuv Sinha, Nicolas Angelard-Gontier, Nan Rosemary Ke, Genevieve Fried, Ryan Lowe, and Joelle Pineau. 2017. Ethical challenges in data-driven dialogue systems. Proceedings of the 2018 AAAI/ACM Conference on AI, Ethics, and Society.

Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, Tom Hennigan, Eric Noland, Katie Millican, George van den Driessche, Bogdan Damoc, Aurelia Guy, Simon Osindero, Karen Simonyan, Erich Elsen, Jack W. Rae, Oriol Vinyals, and Laurent Sifre. 2022. Training compute-optimal large language models.

Minhao Jiang, Ken Ziyu Liu, Ming Zhong, Rylan Schaeffer, Siru Ouyang, Jiawei Han, and Sanmi Koyejo. 2024. Investigating data contamination for pre-training language models.

Nikhil Kandpal, H. Deng, Adam Roberts, Eric Wallace, and Colin Raffel. 2022a. Large language models struggle to learn long-tail knowledge. In International Conference on Machine Learning.

Nikhil Kandpal, Eric Wallace, and Colin Raffel. 2022b. Deduplicating training data mitigates privacy risks in language models.

Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. 2020. Scaling laws for neural language models.

Anjan Karmakar, Julian Aron Prenner, Marco D'Ambros, and Romain Robbes. 2022. Codex hacks hackerrank: Memorization issues and a framework for code synthesis evaluation.

Denis Kocetkov, Raymond Li, Loubna Ben Allal, Jia Li, Chenghao Mou, Carlos Muñoz Ferrandis, Yacine Jernite, Margaret Mitchell, Sean Hughes, Thomas Wolf, Dzmitry Bahdanau, Leandro von Werra, and Harm de Vries. 2022. The stack: 3 tb of permissively licensed source code.

Jooyoung Lee, Thai Le, Jinghui Chen, and Dongwon Lee. 2022. Do language models plagiarize? Proceedings of the ACM Web Conference 2023.

Vladimir I. Levenshtein. 1965. Binary codes capable of correcting deletions, insertions, and reversals. Soviet physics. Doklady, 10:707–710.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. 2023. Starcoder: may the source be with you!

Rien Maertens, Charlotte Van Petegem, Niko Strijbol, Toon Baeyens, Arne Carla Jacobs, Peter Dawyndt, and Bart Mesuere. 2022. Dolos: Language-agnostic plagiarism detection in source code. Journal of Computer Assisted Learning, 38(4):1046–1061.

Inbal Magar and Roy Schwartz. 2022. Data contamination: From memorization to exploitation. In Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers), pages 157–165, Dublin, Ireland. Association for Computational Linguistics.

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.

Ansong Ni, Pengcheng Yin, Yilun Zhao, Martin Riddell, Troy Feng, Rui Shen, Stephen Yin, Ye Liu, Semih Yavuz, Caiming Xiong, et al. 2023. L2ceval: Evaluating language-to-code generation capabilities of large language models. arXiv preprint arXiv:2309.17446.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2023. Codegen: An open large language model for code with multi-turn program synthesis.

Yonatan Oren and Nicole Meister. 2023. Proving test set contamination in black box language models.

Zhen Peng, Zhizhi Wang, and Dong Deng. 2023. Near-duplicate sequence search at scale for large language model memorization evaluation. Proceedings of the ACM on Management of Data, 1:1 – 18.

Lutz Prechelt and Guido Malpohl. 2003. Finding plagiarisms among a set of programs with jplag. Journal of Universal Computer Science, 8.

Federico Ranaldi, Elena Sofia Ruzzetti, Dario Onorati, Leonardo Ranaldi, Cristina Giannone, Andrea Favalli, Raniero Romagnoli, and Fabio Massimo Zanzotto. 2024. Investigating the impact of data contamination of large language models in text-to-sql translation.

Yasaman Razeghi, Robert L Logan IV, Matt Gardner, and Sameer Singh. 2022. Impact of pretraining term frequencies on few-shot numerical reasoning. In Findings of the Association for Computational Linguistics: EMNLP 2022, pages 840–854, Abu Dhabi, United Arab Emirates. Association for Computational Linguistics.

Sandip Sarkar, Dipankar Das, Partha Pakray, and Alexander Gelbukh. 2016. JUNITMZ at SemEval-2016 task 1: Identifying semantic similarity using Levenshtein ratio. In Proceedings of the 10th International Workshop on Semantic Evaluation (SemEval-2016), pages 702–705, San Diego, California. Association for Computational Linguistics.

Weijia Shi, Anirudh Ajith, Mengzhou Xia, Yangsibo Huang, Daogao Liu, Terra Blevins, Danqi Chen, and Luke Zettlemoyer. 2023. Detecting pretraining data from large language models.

Om Thakkar, Swaroop Indra Ramaswamy, Rajiv Mathews, and Franccoise Beaufays. 2020. Understanding unintended memorization in federated learning. ArXiv, abs/2006.07490.

Aleena Anna Thomas, David Ifeoluwa Adelani, Ali Davody, Aditya Mogadala, and Dietrich Klakow. 2020. Investigating the impact of pre-trained word embeddings on memorization in neural networks. In Workshop on Time-Delay Systems.

Shuo Yang, Wei-Lin Chiang, Lianmin Zheng, Joseph E. Gonzalez, and Ion Stoica. 2023. Rethinking benchmark and contamination for language models with rephrased samples.

Zhiyuan Yu, Yuhao Wu, Ning Zhang, Chenguang Wang, Yevgeniy Vorobeychik, and Chaowei Xiao. 2023. CodeIPPrompt: Intellectual property infringement assessment of code language models. In Proceedings of the 40th International Conference on Machine Learning, volume 202 of Proceedings of Machine Learning Research, pages 40373–40389. PMLR.

A Additional Examples and Results

A.1 All Model Series

In Tab. 2 we present the results on the largest versions of the StarCoder-Base, Pythia and CodeGen-nl model series. In Tab. 4 we show results on four model versions from each model series.

A.2 Relevant Info for Models on the HumanEval Benchmark

We provide versions of Fig. 3 and Fig. 6 for the HumanEval benchmark in Fig. 11 and Fig. 12 respectively.

A.3 Examples of Similarity Scores

In Fig. 9 and Fig. 10 we provide examples of programs found within the training data and the relevant similarity score returned for them. A similarity score of 70 typically represents a program that is no longer similar to the gold program.

```
Prompt: Write a python function to count
    positive numbers in a list.
```

(a) Problem Description

```
def pos_count(list):
  pos_count= 0
  for num in list:
    if num >= 0:
      pos_count += 1
  return pos_count
```

(b) Gold Program

```
def num_sym_points(sv):
    count = 0
    for v in sv:
        if v <= 0:
            count += 1

    return count
```

(c) Matched program in STARCODERDATA. Semantic-level similarity score = 100

```
def get_degree(d):
    count = 0
    for key in d:
        if d[key] > 0:
            count += 1
    return count
```

(d) Matched program in STARCODERDATA. Semantic-level similarity score = 72.73

Figure 9: Examples of different programs and their corresponding Dolos scores when compared to a gold program from the MBPP benchmark.

```
Prompt: Write a function that takes two
    lists and returns true if they have
    at least one common element.
```

(a) Problem Description

```
def common_element(list1, list2):
    result = False
    for x in list1:
        for y in list2:
            if x == y:
                result = True
                return result
```

(b) Gold Program

```
atches

def compare_lists(list1,list2):
    result = False
    for x in list1:
        for y in list2:
            if x == y:
                result = True
                return result
```

(c) Matched program in StarCoderData. Surface-level similarity score = 92

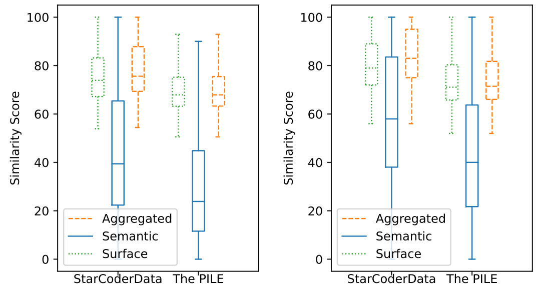
```
#def two_data(list1, list2):
#     result = False
 #       for x in list2:
  #           if i == x:
   #               result = True
      #           return result
#print(two_data([3,4,
```

(d) Matched program in StarCoderData. Surface-level similarity score = 81

```
def top_ingredients(self, n):
        res = {}
        for a in self.items:
            for i in a.ingredients:
                try:
                    res[i] += 1
                exce
```

(e) Matched program in StarCoderData. Surface-level similarity score = 70

Figure 10: Examples of different programs and their corresponding Levenshtein scores when compared to a gold program from the MBPP benchmark.



(a) Top-10 Scores. (b) Top-1 Score.

Figure 11: We show the similarity scores for both The Pile and StarCoderData found by searching for answers to the gold programs in the HumanEval benchmark. We compare the similarity scores from different techniques, as well as the difference between using the top-1 score and the top-10 scores.

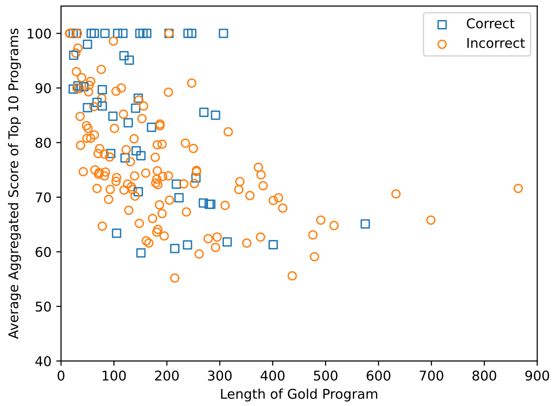


Figure 12: Length of gold programs in the HumanEval benchmark plotted against the average aggregated similarity score of the top-10 scores within the training dataset used for the StarCoderBase model family.

A.4 Perfect Matches

We provide lists of every question that was seen 10 or more times during training for models trained on a specific dataset. These can be found in Tab. 5 through Tab. 13.

Models	MBPP			HumanEval		
	Acc_o	$\Uparrow^{10\%}$	$\Downarrow_{10\%}$ ($\Delta_{\Updownarrow}$)	Acc_o	$\Uparrow^{10\%}$	$\Downarrow_{10\%}$ ($\Delta_{\Updownarrow}$)
StarCoderBase-1B	23.4	54.0	4.0 (-50.0)	16.5	56.6	18.8 (-37.5)
StarCoderBase-3B	29.8	64.0	8.0 (-56.0)	22.0	75.0	25.0 (-50.0)
StarCoderBase-7B	37.2	70.0	18.0 (-52.0)	31.3	68.8	31.3 (-37.5)
StarCoderBase-15.5B	41.6	72.0	22.0 (-50.0)	30.5	75.0	31.3 (-43.7)
Pythia-1.4B	4.4	18.0	0.0 (-18.0)	4.9	25.0	0.0 (-25.0)
Pythia-2.8B	12.0	30.0	2.0 (-28.0)	7.3	31.3	0.0 (-31.3)
Pythia-6.9B	12.6	34.0	2.0 (-32.0)	6.7	43.8	0.0 (-43.8)
Pythia-12B	17.8	40.0	8.0 (-32.0)	9.8	56.3	0.0 (-56.3)
CodeGen-NL-350M	2.2	8.0	0.0 (-8.0)	3.0	12.5	0.0 (-12.5)
CodeGen-NL-2B	12.7	36.0	4.0 (-32.0)	7.9	37.5	0.0 (-37.5)
CodeGen-NL-6B	15.8	42.0	6.0 (-36.0)	8.5	37.5	0.0 (-37.5)
CodeGen-NL-16B	19.6	48.0	6.0 (-42.0)	14.6	62.5	0.0 (-62.5)

Table 4: We show the performance gap ($\Delta_{\Updownarrow}$) between the top 10% ($\Uparrow^{10\%}$) and bottom 10% ($\Downarrow_{10\%}$) of questions for the MBPP and HumanEval benchmarks compared against "Acc_o" the original model accuracy. This is the full version of Tab. 2, showing results for four models in each model series.

Natural Language Question	Gold Program	Found 100% Matches
`Write a function to find the closest smaller number than n.`	`def closest_num(N):` `  return (N - 1)`	`def parent(i):` `    return (i - 1)`
`Write a python function to count true booleans in the given list.`	`def count(lst):` `    return sum(lst)`	`def average(lst):` `    return sum(lst)`
`Write a python function to find smallest number in a list.`	`def smallest_num(xs):` `  return min(xs)`	`def min_usecase3(x):` `    return min(x)`

Table 5: All questions flagged as being seen by models trained on the PILE 10 or more times within the MBPP benchmark

Natural Language Question	Gold Program	Found 100% Matches
Write a function to find the n-th rectangular number.	`def find_rect_num(n):` `  return n*(n + 1)`	`def get_sum(n):` `    return n * (n + 1)`
Write a python function to find the last digit of a given number.	`def last_Digit(n) :` `    return (n % 10)`	`def shift_right(b):` `    return (b << 1)`
Write a function to find the closest smaller number than n.	`def closest_num(N):` `  return (N - 1)`	`def percentage(x):` `  return (x - 1)`
Write a python function to find smallest number in a list.	`def smallest_num(xs):` `  return min(xs)`	`def smallest(l):` `    return min(l)`

Write a python function to count positive numbers in a list.	`def pos_count(list):`
	`  pos_count= 0`
	`  for num in list:`
	`    if num >= 0:`
	`      pos_count += 1`
	`  return pos_count`
		`def array_count9(nums):`
		`    count = 0`
		`    for num in nums:`
		`      if num == 9:`
		`        count += 1`
		`    return count`

Write a function to swap two numbers.	`def swap_numbers(a,b):`
	` temp = a`
	` a = b`
	` b = temp`
	` return (a,b)`
		`def swap(a,b):`
		`    tmp = a`
		`    a = b`
		`    b = tmp`
		`    return a,b`

[link text](https:// [link text](https:// [link text](https://))) write a function to convert a string to a list.	`def string_to_list(string):`
	`    lst = list(string.split(" "))`
	`    return lst`
		`def String_to_list (Strings):`
		`    list1=list(Strings.split(" "))`
		`    return l`

Write a python function to find the minimum of two numbers.	`def minimum(a,b):`
	`    if a <= b:`
	`        return a`
	`    else:`
	`        return b`
		`def minimum(a, b):`
		`    if a <= b:`
		`        return a`
		`    else:`
		`        return b`
		
		`def`

Write a function to check whether an element exists within a tuple.	`def check_tuplex(tuplex,tuple1):`
	`  if tuple1 in tuplex:`
	`    return True`
	`  else:`
	`     return False`
		`def check_for_tag(ele, tag):`
		`    if tag in ele:`
		`        return True`
		`    else:`
		`        return False`

Write a python function to count true booleans in the given list.	`def count(lst):` `    return sum(lst)`	`def mean(lst):` `        return sum(lst)`

Table 6: All questions flagged as being seen by models trained on STARCODERDATA 10 or more times within the MBPP benchmark (Part 1)

Natural Language Question	Gold Program	Found 100% Matches

Write a python function to find the largest negative number from the given list.	def largest_neg(list1):
	max = list1[0]
	for x in list1:
	if x < max :
	max = x
	return max
		def minimum(list):
		min = list[0]
		for i in list:
		if i < min:
		min = i
		return min

Write a function to convert the given decimal number to its binary equivalent.	def decimal_to_binary(n): return bin(n).replace("0b","")	def decimal_to_binary(n): return bin(n). replace("0b", "")

Write a python function to find the maximum of two numbers.	def maximum(a,b):
	if a >= b:
	return a
	else:
	return b
		def maximum(a, b):
		
		if a >= b:
		return a
		else:
		return b

Write a function to extract every specified element from a given two dimensional list.	def specified_element(nums, N): result = [i[N] for i in nums] return result	def _get_mean(names, table): x = [table[name] for name in names] return su
Write a function to find the nth hexagonal number.	def hexagonal_num(n): return n*(2*n - 1)	def hexagonal(n): return n * (2*n - 1)
Write a function to remove all elements from a given list present in another list.	def remove_elements(list1, list2): result = [x for x in list1 if x not in list2] return result	def intersection(lst1, lst2): lst3 = [value for value in lst1 if value not in lst2] return l

Write a function to calculate the sum of the positive integers of n+(n-2)+(n-4)... (until n-x =< 0).	def sum_series(n):
	if n < 1:
	return 0
	else:
	return n + sum_series(n - 2)
		def sum_series(n):
		if n < 1:
		return 0
		else:
		return n + sum_series(n - 2)

Table 7: All questions flagged as being seen by models trained on STARCODERDATA 10 or more times within the MBPP benchmark (Part 2)

Natural Language Question	Gold Program	Found 100% Matches
`from typing import List def filter_by_prefix(strings: List[str], prefix: str) -> List[str]: """ Filter an input list of strings only for ones that start with a given prefix. >>> filter_by_prefix([], 'a') [] >>> filter_by_prefix(['abc', 'bcd', 'cde', 'array'], 'a') ['abc', 'array'] """`	`return [x for x in strings if substring in x]`	`return [tag for tag in tags if substr in tag]`

`def greatest_common_divisor(a: int, b: int) -> int: """ Return a greatest common divisor of two integers a and b >>> greatest_common_divisor(3, 5) 1 >>> greatest_common_divisor(25, 15) 5 """`	`while b:`
	`    a, b = b, a % b`
	`return a`
		`while b:`
		`    a, b = b, a % b`
		`return a`

`from typing import List def concatenate(strings: List[str]) -> str: """ Concatenate list of strings into a single string >>> concatenate([]) '' >>> concatenate(['a', 'b', 'c']) 'abc' """`	`return ''.join(strings)`	`return ''.join(strings)`
`from typing import List def filter_by_prefix(strings: List[str], prefix: str) -> List[str]: """ Filter an input list of strings only for ones that start with a given prefix. >>> filter_by_prefix([], 'a') [] >>> filter_by_prefix(['abc', 'bcd', 'cde', 'array'], 'a') ['abc', 'array'] """`	`return [x for x in strings if x.startswith(prefix)]`	`return [i for i in tests if i.startswith(prefix)]`
`def encode_cyclic(s: str): """ returns encoded string by cycling groups of three characters. """ # split string to groups. Each of length 3. groups = [s[(3 * i):min((3 * i + 3), len(s))] for i in range((len(s) + 2) // 3)] # cycle elements in each group. Unless group has fewer elements than 3. groups = [(group[1:] + group[0]) if len(group) == 3 else group for group in groups] return "".join(groups) def decode_cyclic(s: str): """ takes as input string encoded with encode_cyclic function. Returns decoded string. """`	`return encode_cyclic(encode_cyclic(s))`	`return encodedValue(encode(value));`
`def add(x: int, y: int): """Add two numbers x and y >>> add(2, 3) 5 >>> add(5, 7) 12 """`	`return x + y`	`return x + y`

Table 8: All questions flagged as being seen by models trained on the PILE 10 or more times within the HumanEval benchmark (Part 1)

Natural Language Question	**Gold Program**	**Found 100% Matches**
def same_chars(s0: str, s1: str): """ Check if two words have the same characters. >>> same_chars(’eabcdzzzz’, ’dddzzzzzzzddeddabc’) True >>> same_chars(’abcd’, ’dddddddabc’) True >>> same_chars(’dddddddabc’, ’abcd’) True >>> same_chars(’eabcd’, ’dddddddabc’) False >>> same_chars(’abcd’, ’dddddddabce’) False >>> same_chars(’eabcdzzzz’, ’dddzzzzzzzddddabc’) False """	return set(s0) == set(s1)	return set(l1) == set(l2)
def multiply(a, b): """Complete the function that takes two integers and returns the product of their unit digits. Assume the input is always valid. Examples: multiply(148, 412) should return 16. multiply(19, 28) should return 72. multiply(2020, 1851) should return 0. multiply(14,-15) should return 20. """	return abs(a % 10) * abs(b % 10)	return abs(fa - f0) < abs(fb - f0)

Table 9: All questions flagged as being seen by models trained on the PILE 10 or more times within the HumanEval benchmark (Part 2)

Natural Language Question	Gold Program	Found 100% Matches
def strlen(string: str) -> int: """ Return length of given string >>> strlen(’’) 0 >>> strlen(’abc ’) 3 """	return len(string)	return len(string)
def flip_case(string: str) -> str: """ For a given string, flip lowercase characters to uppercase and uppercase to lowercase. >>> flip_case(’Hello ’) ’hELLO’ """	return string.swapcase()	return string_. swapcase()
def sort_third(l: list): """This function takes a list l and returns a list l’ such that l’ is identical to l in the indicies that are not divisible by three, while its values at the indicies that are divisible by three are equal to the values of the corresponding indicies of l, but sorted. >>> sort_third ([1, 2, 3]) [1, 2, 3] >>> sort_third([5, 6, 3, 4, 8, 9, 2]) [2, 6, 3, 4, 8, 9, 5] """	l = list(l) l[::3] = sorted(l[::3]) return l	l = list(str(n)) s = sorted(l) return s != l

def fizz_buzz(n: int): """Return the
number of times the digit 7
appears in integers less than n
which are divisible by 11 or 13.
>>> fizz_buzz(50) 0 >>>
fizz_buzz(78) 2 >>> fizz_buzz
(79) 3 """
	ns = []
	for i in range(n):
	if i % 11 == 0 or i % 13 ==
	0:
	ns.append(i)
	s = ’’.join(list(map(str, ns)))
	ans = 0
	for c in s:
	ans += (c == ’7’)
	return ans
		fizzy = []
		for i in range(1, n
		+1):
		if i % 3 == 0
		and i % 5 == 0:
		fizzy.
		append(’FizzBuzz’)
		elif i % 3 ==
		0:
		fizzy.
		append(’Fizz’)
		elif i % 5 ==
		0:

from typing import List, Optional def
longest(strings: List[str]) ->
Optional[str]: """ 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’ """
	if not strings:
	return None
	
	maxlen = max(len(x) for x in
	strings)
	for s in strings:
	if len(s) == maxlen:
	return s
		if not strings:
		return ’’
		prefix = strings[0]
		for s in strings:
		if len(s) < len
		(prefix):
		prefix =
		prefix[:len(s

def how_many_times(string: str, substring: str) -> int: """ Find how many times a given substring can be found in the original string. Count overlaping cases. >>> how_many_times(’’, ’a’) 0 >>> how_many_times(’aaa’, ’a’) 3 >>> how_many_times(’aaaa’, ’aa’) 3 """	times = 0 for i in range(len(string) - len(substring) + 1): if string[i:i+len(substring)] == substring: times += 1 return times	result = 0 for i in range(len(string) - len(sub_string) + 1): if string[i:i+ len(sub_string)] == sub_string: result += 1 return res

Table 10: All questions flagged as being seen by models trained on STARCODERDATA 10 or more times within the HumanEval benchmark (Part 1)

Natural Language Question	Gold Program	Found 100% Matches

from typing import List def sort_numbers(numbers: str) -> str: """ Input is a space-delimited string of numberals from 'zero' to 'nine'. Valid choices are 'zero', 'one', 'two', 'three', 'four', 'five', 'six', 'seven', 'eight' and 'nine'. Return the string with numbers sorted from smallest to largest >>> sort_numbers('three one five') 'one three five' """	value_map = {
	'zero': 0,
	'one': 1,
	'two': 2,
	'three': 3,
	'four': 4,
	'five': 5,
	'six': 6,
	'seven': 7,
	'eight': 8,
	'nine': 9
	}
	return ' '.join(sorted([x for x in numbers.split(' ') if x], key=lambda x: value_map[x]))
		nine
		number_map = {
		'zero': 0,
		'one': 1,
		'two': 2,
		'three': 3,
		'four': 4,
		'five': 5,
		'six': 6,
		'seven': 7,
		'eight': 8,
		'nine': 9
		}
		
		# Override the init method and use the kwarg "name" to set a string that will be sl

def tri(n): """Everyone knows Fibonacci sequence, it was studied deeply by mathematicians in the last couple centuries. However, what people don't know is Tribonacci sequence. Tribonacci sequence is defined by the recurrence: tri(1) = 3 tri(n) = 1 + n / 2, if n is even. tri(n) = tri(n - 1) + tri(n - 2) + tri(n + 1), if n is odd. For example: tri(2) = 1 + (2 / 2) = 2 tri(4) = 3 tri(3) = tri(2) + tri(1) + tri(4) = 2 + 3 + 3 = 8 You are given a non-negative integer number n, you have to a return a list of the first n + 1 numbers of the Tribonacci sequence. Examples: tri(3) = [1, 3, 2, 8] """	if n == 0:
	return [1]
	my_tri = [1, 3]
	for i in range(2, n + 1):
	if i % 2 == 0:
	my_tri.append(i / 2 + 1)
	else:
	my_tri.append(my_tri[i - 1] + my_tri[i - 2] + (i + 3) / 2)
	return my_tri
		= len(my_list)
		if list_len == 0:
		return None
		
		bool_list = []
		for i in range(list_len):
		if my_list[i] % 2 == 0:
		bool_list.append(True)
		else:
		bool_list.append(False)
		return (bool_list)

def check_if_last_char_is_a_letter(txt): ''' Create a function that returns True if the last character of a given string is an alphabetical character and is not a part of a word, and False otherwise. Note: "word" is a group of characters separated by space. Examples: check_if_last_char_is_a_letter("apple pie") -> False check_if_last_char_is_a_letter("apple pi e") -> True check_if_last_char_is_a_letter("apple pi e ") -> False check_if_last_char_is_a_letter("") -> False '''	check = txt.split(' ')[-1] return True if len(check) == 1 and (97 <= ord(check.lower()) <= 122) else False	prefix = re.split(r'[\._]', id)[0] return True if len(prefix) == 6 and int(prefix) > 101000 else False

def can_arrange(arr): """Create a function which returns the largest index of an element which is not greater than or equal to the element immediately preceding it. If no such element exists then return -1. The given array will not contain duplicate values. Examples: can_arrange([1,2,4,3,5]) = 3 can_arrange([1,2,3]) = -1 """	ind=-1
	i=1
	while i<len(arr):
	if arr[i]<arr[i-1]:
	ind=i
	i+=1
	return ind
		i = 1
		while i < len(arr):
		if arr[i-1] < arr[i]:
		i += 1
		elif arr[i

Table 11: All questions flagged as being seen by models trained on STARCODERDATA 10 or more times within the HumanEval benchmark (Part 2)

Natural Language Question	Gold Program	Found 100% Matches
def is_equal_to_sum_even(n): """ Evaluate whether the given number n can be written as the sum of exactly 4 positive even numbers Example is_equal_to_sum_even(4) == False is_equal_to_sum_even(6) == False is_equal_to_sum_even(8) == True """	return n%2 == 0 and n >= 8	5 return n2 == 7 and n1 >=
def car_race_collision(n: int): """ Imagine a road that’s a perfectly straight infinitely long line. n cars are driving left to right; simultaneously, a different set of n cars are driving right to left. The two sets of cars start out being very far from each other. All cars move in the same speed. Two cars are said to collide when a car that’s moving left to right hits a car that’s moving right to left. However, the cars are infinitely sturdy and strong; as a result, they continue moving in their trajectory as if they did not collide. This function outputs the number of such collisions. """	return n**2	return n**2
def reverse_delete(s,c): """Task We are given two strings s and c, you have to deleted all the characters in s that are equal to any character in c then check if the result string is palindrome. A string is called palindrome if it reads the same backward as forward. You should return a tuple containing the result string and True/False for the check. Example For s = "abcde", c = "ae", the result should be (’bcd’,False) For s = "abcdef", c = "b" the result should be (’acdef’,False) For s = "abcdedcba", c = "ab", the result should be (’cdedc’,True) """	s = ’’.join([char for char in s if char not in c]) return (s,s[::-1] == s)	) s = ’’.join([char for char in s if char not in dashes]) return s.lower()

def minSubArraySum(nums): """ Given an array of integers nums, find the minimum sum of any non-empty sub-array of nums. Example minSubArraySum([2, 3, 4, 1, 2, 4]) == 1 minSubArraySum([-1, -2, -3]) == -6 """	max_sum = 0
	s = 0
	for num in nums:
	    s += -num
	    if (s < 0):
	        s = 0
	    max_sum = max(s, max_sum)
	if max_sum == 0:
	    max_sum = max(-i for i in nums)
	min_sum = -max_sum
	return min_sum
		max_sum, sub_sum = nums[0], nums[0]
		for num in nums[1:]:
		    s = num
		    if sub_sum > 0:
		        s += sub_sum
		    max_sum = max(s, max_sum)
		    sub_sum = s
		return max_sum

Table 12: All questions flagged as being seen by models trained on STARCODERDATA 10 or more times within the HumanEval benchmark (Part 3)

<table>
<tr><th>Natural Language Question</th><th>Gold Program</th><th>Found 100% Matches</th></tr>
<tr><td>def max_fill(grid, capacity): import math """ You are given a rectangular grid of wells. Each row represents a single well, and each 1 in a row represents a single unit of water. Each well has a corresponding bucket that can be used to extract water from it, and all buckets have the same capacity. Your task is to use the buckets to empty the wells. Output the number of times you need to lower the buckets. Example 1: Input: grid : [[0,0,1,0], [0,1,0,0], [1,1,1,1]] bucket_capacity : 1 Output: 6 Example 2: Input: grid : [[0,0,1,1], [0,0,0,0], [1,1,1,1], [0,1,1,1]] bucket_capacity : 2 Output: 5 Example 3: Input: grid : [[0,0,0], [0,0,0]] bucket_capacity : 5 Output: 0 Constraints: * all wells have the same length * 1 <= grid. length <= 10^2 * 1 <= grid[:,1]. length <= 10^2 * grid[i][j] -> 0 | 1 * 1 <= capacity <= 10 """</td><td>return sum([math.ceil(sum(arr)/ capacity) for arr in grid])</td><td>return sum(int(math.ceil(ntasks)) for ntasks in rows)</td></tr>
<tr><td>def add(x: int, y: int): """Add two numbers x and y >>> add(2, 3) 5 >>> add(5, 7) 12 """</td><td>return x + y</td><td>return x + y</td></tr>
<tr><td>def do_algebra(operator, operand): """ Given two lists operator, and operand. The first list has basic algebra operations, and the second list is a list of integers. Use the two given lists to build the algebric expression and return the evaluation of this expression. The basic algebra operations: Addition (+) Subtraction (-) Multiplication (*) Floor division (//) Exponentiation (**) Example: operator['+', '*', '-'] array = [2, 3, 4, 5] result = 2 + 3 * 4 - 5 => result = 9 Note: The length of operator list is equal to the length of operand list minus one . Operand is a list of of non-negative integers. Operator list has at least one operator, and operand list has at least two operands. """</td><td>expression = str(operand[0])
for oprt, oprn in zip(operator, operand[1:]):
    expression+= oprt + str(oprn)
return eval(expression)</td><td>result = str(self. operand[0])
for operator, operand in zip(self.operator, self.operand[1:]):
  result += ' {} {}'.format(operator, operan</td></tr>
<tr><td>def encode(message): """ Write a function that takes a message, and encodes in such a way that it swaps case of all letters, replaces all vowels in the message with the letter that appears 2 places ahead of that vowel in the english alphabet. Assume only letters. Examples: >>> encode('test') 'TGST' >>> encode('This is a message') ' tHKS KS C MGSSCGG' """</td><td>vowels = "aeiouAEIOU"
vowels_replace = dict([(i, chr(ord(i) + 2)) for i in vowels])
message = message.swapcase()
return ''.join([vowels_replace[i] if i in vowels else i for i in message])</td><td>vovels = " aeiouAEIOU"
    stack = []
    for ch in s:
        if ch in vovels:
            stack. append(ch)
    result = []
    for i in s:
        if i in vovels</td></tr>
</table>

Table 13: All questions flagged as being seen by models trained on StarCoderData 10 or more times within the HumanEval benchmark (Part 4)