Are Large Language Models Memorizing Bug Benchmarks?

Daniel Ramos
Carnegie Mellon University, INESC-ID
Pittsburgh, PA, USA
danielrr@cmu.edu

Claudia Mamede*
Carnegie Mellon University, FEUP
Pittsburgh, PA, USA
cmamede@andrew.cmu.edu

Kush Jain*
Carnegie Mellon University
Pittsburgh, PA, USA
kdjain@andrew.cmu.edu

Paulo Canelas*
Carnegie Mellon University, LASIGE
Pittsburgh, PA, USA
pasantos@andrew.cmu.edu

Catarina Gamboa*
Carnegie Mellon University, LASIGE
Pittsburgh, PA, USA
cgamboa@andrew.cmu.edu

Claire Le Goues
Carnegie Mellon University
Pittsburgh, PA, USA
clegoues@andrew.cmu.edu

***Abstract*—Large Language Models (LLMs) have become integral to various software engineering tasks, including code generation, bug detection, and repair. To evaluate model performance in these domains, numerous bug benchmarks containing real-world bugs from software projects have been developed. However, a growing concern within the software engineering community is that these benchmarks may not reliably reflect true LLM performance due to the risk of data leakage. Despite this concern, limited research has been conducted to quantify the impact of potential leakage.**

In this paper, we systematically evaluate popular LLMs to assess their susceptibility to data leakage from widely used bug benchmarks. To identify potential leakage, we use multiple metrics, including a study of benchmark membership within commonly used training datasets, as well as analyses of negative log-likelihood and `5-gram accuracy`. Our findings show that certain models, in particular `codegen-multi`, exhibit significant evidence of memorization in widely used benchmarks like `Defects4J`, while newer models trained on larger datasets like `LLaMa 3.1` exhibit limited signs of leakage. These results highlight the need for careful benchmark selection and the adoption of robust metrics to adequately assess models capabilities.

***Index Terms*—Automated Program Repair, Large Language Model, Data Leakage**

I. Introduction

Large language models (LLMs) have become ubiquitous for various software engineering tasks. Assessing these models' abilities in context, beyond the basic evaluations typically performed upon release (e.g., on HumanEval [1]), benefits from realistic benchmarks that represent real-world software development tasks. Two significant such tasks are bug finding, through automated fault localization (FL) [2]; and bug fixing, through automated program repair [3] (APR). The Software Engineering community has released numerous *bug benchmarks* for evaluating success on these tasks, consisting of real bugs from open-source software projects. Notable such datasets include, for example, `Defects4J` [4] (`Java`) and `BugsInPy` [5] (`Python`); similarly, ML researchers recently introduced `SWEBench` [6].

*Equal contribution

However, a growing concern in software engineering research is the degree to which data leakagecompromises the evaluation of true model capability [7], [8]. *Data leakage* refers to the use of information during model training that would not normally be available during prediction, leading to inflated performance metrics that misrepresent a model's true effectiveness. The programs and bugs in many benchmarks and solutions have been publicly accessible for years, increasing the chance they were incorporated into LLM training data. For instance, the widely-used `Defects4J` dataset, based on popular `Java` projects, was first released in 2014.

To illustrate, consider the example shown in Figure 1. Here, we prompted `codegen-multi` (6 billion parameters) to predict the next tokens based on an incomplete snippet (highlighted in yellow) from the bug-fix file for `Defects4J` bug #39 in the `Lang` project. The generated text, highlighted in blue, shows that `codegen-multi` reproduced the remainder of the solution file verbatim, including specific comments (lines 31 and 35), token by token. This exact reproduction strongly suggests that the model has memorized the benchmark solution. Such behavior underscores the need for caution when using benchmarks like `Defects4J`, as they may lead to misleading conclusions about a model's generalizability.

Researchers in other domains, like NLP or ML, have attempted to quantify data leakage in their datasets [9], [10]. To the best of our knowledge, this effort has not been undertaken for popular bug datasets in Software Engineering. Detecting leakage is challenging. LLMs are often pre-trained by organizations that do not disclose their datasets, the total volume of data, nor model parameters. Consequently, determining whether a benchmark has been directly included or merely mirrored in the model's training set is difficult. Even verifying the presence of data only establishes that the model has seen it, not necessarily that it has memorized it, particularly when it is one training sample among billions.

Thus, we ask: **Are large language models memorizing bug benchmarks?** We systematically evaluate popular LLMs to quantify their susceptibility to data leakage on widely used

```
1                 * * ommited for brevity * *
2  // no search strings found, we are done
3  if (textIndex == -1) {
4    return text;
5  }
6  int start = 0;
7  // get a good guess on the size of the result buffer so
8  // it doesnt have to double if it goes over a bit
9  int increase = 0;
10 // count the replacement text elements that are larger
11 // than their corresponding text being replaced
12 for (int i = 0; i < searchList.length; i++) {
13   if (searchList[i]==null || replacementList[i]==null) {
14      continue;
15   }
16   int greater = replacementList[i].length() -
17                 searchList[i].length();
18   if (greater > 0) {
19     increase += 3 * greater; // assume 3 matches
20   }
21 }
22 //have upper-bound at 20% increase, then let Java take...
```

patch

Prompt input to `codegen-multi`.

Both `codegen-multi` output and the `Defects4J` solution.

Fig. 1: Excerpt from `Defects4J` (`Lang:Bug 39`). Given the first lines of the function until line 11, `codegen-multi` generated lines 12 to 23, matching the benchmark solution.

bug benchmarks and raise awareness of the risks associated with using established benchmarks, which may inadvertently inflate performance by testing memorized data.

We use multiple metrics to detect potential leakage. First, we investigate whether benchmark data has *membership* within `TheStack`, a widely-used used pretraining code dataset. Following this, we apply two core metrics for leakage from prior work [9], [10]: *Negative Log-Likelihood (`NLL`)* and `5-gram accuracy`. `NLL` provides insight into model familiarity with code snippets; `5-gram accuracy` assesses the model's ability to reproduce exact sequences. We apply these metrics to both well-known bug benchmarks, and a new dataset of high-quality code repositories from 2024, which we mined from `GitHub`. This new dataset is less likely to have appeared in models' training, which allows us to compare model performance between potentially familiar data versus likely novel code.

Our findings suggest that older models, in particular `codegen-multi`, exhibit very high `5-gram accuracy` and low `NLL` on benchmark data, indicating a higher likelihood of memorization. Our evidence suggests that newer models trained on more extensive datasets, like `LLaMa 3.1`, show less memorization. Nonetheless, across all metrics, and models, `Defects4J` — arguably the most widely-used bug benchmark — consistently exhibits the highest rate of potential memorization. These results underscore the importance of carefully selecting benchmarks to ensure reliable evaluations.

TABLE I: Evaluation benchmark statistics, including newer and older benchmarks to assess leakage reduction over time.

Benchmark	Year	# Bugs	LOC (k)	# Stars	Language
Defects4J (v1.5)	2019	438	321	736	Java
BugsInPy	2020	493	1253	80	Python
BugsC++	2021	209	4297	42	C++
GitBug-Java	2023	199	257	26	Java
SWEBenchLite	2023	300	3148	1954	Python
New Java Repos	2024	-	132	>100	Java
New Python Repos	2024	-	65	>100	Python

II. Methodology

Figure 2 overviews our methodology, which comprises three major components: (1) data collection (Section II-A, (2) model selection (Section II-B), and (3) evaluation (Section III).

A. Data Collection & Filtering.

We select widely used bug benchmarks across common programming languages, gathering ground-truth files containing reference solutions for each bug fix. To provide a likely not-leaked datasets for comparison, we also curated a set of recent open-source repositories from `GitHub`.

To collect benchmarks of interest, we reviewed program-repair.org and selected highly starred benchmarks across three programming languages: `BugsCpp` [11], `Defects4J` [4], and `BugsInPy` [5]. We next included two recent datasets to serve as reference points: `GitBug-Java` [12], which was recently published to address the leakage issue, and `SWEBench-Lite` [6], due to its rising popularity for code-related tasks [13], [14], [15]. Table I shows details, including release year, bug count, lines of code, stars, and language. We choose `SWEBench-Lite` instead of `SWEBench` due to computational constraints.

We collected the ground truth files for each patched bug. To reduce computational costs and prevent bias towards particular files, we removed files with more than 85% overlap to produce a unique sample set. Duplicate files appear when multiple bugs in a dataset occur in the same file (e.g., `Defects4J` in project `Lang`, Bugs 1 and 3 affect the same file). We kept the oldest file for consistency. We collected only fixed files since these correspond to the solutions a model may have memorized.

We collected a new dataset of 3,214 `GitHub` repositories written in `Java` (1,183) and `Python` (2,031), targeting repositories from 2024 to reduce the likelihood that state-of-the-art LLMs have seen them. To focus on high-quality repositories, we applied a 100-star minimum threshold as a proxy for community interest. To ensure novelty, we used MinHash [16] with Locality Sensitive Hashing (LSH) [17] to filter repositories overlapping with existing training data. Due to code duplication, some files in 2024 repositories might overlap with older repositories. Therefore, we included repositories from 2022–2023 (>100 stars) to exclude 2024 repositories containing duplicated data. Finally, we randomly sampled 250 files per language to manage computational resources for evaluation.

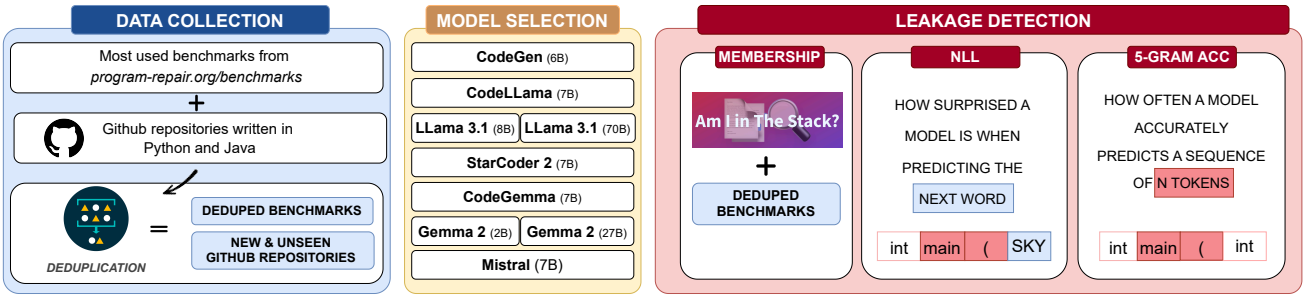


Fig. 2: Overview of our methodology for detecting leakage. We collected bug benchmarks and unseen repositories from 2024. We evaluated `NLL` and N-gram accuracy on base models, and analyzed membership of the benchmarks in `TheStack`.

TABLE II: Models used for evaluation, including their training budget in trillions of tokens, number of layers, and cutoff year.

Model	Tokens (T)	Cutoff Year
Codegen Multi (6B) [18]	1	2022
CodeLlama (7B) [19]	2.5	2023
LlaMa 3.1 (8B / 70B) [20]	15.0	2024
StarCoder 2 (7B) [21]	3.5	2024
Gemma 2 (2B / 27B) [22]	2.0 / 13.0	2024
CodeGemma (7B) [23]	6.5	2024
Mistral (7B) [24]	-	-

B. Model Selection.

We select a combination of models used for fault localization [25], program repair [26], and vulnerability detection [27]. Table II shows model information.These models are from families of well-known code-related models, including `codegen-multi`, `LLaMa 3.1`, `Gemma 2`, `StarCoder`, and `Mistral`.

Note that we focus on open-source base models because our method requires computing the negative log-likelihood (`NLL`) on sequences, which is generally not possible with closed-source models. We exclude instruction-tuned models and concentrate solely on pretrained models before fine-tuning. Since instruction-tuned models are optimized for conversational formats, `n-gram accuracy` may be a less suitable metric for measuring memorization in these models.

C. Leakage Detection

We follow strategies from prior work [9], [10] to evaluate models for potential data leakage. *Membership* operates at the repository level; *Negative Log Likelihood* and *N-gram accuracy*, at the file level (i.e., the compute model familiarity with a given file). In the bug datasets, these are the fixed (patched) files; in our novel dataset, they are randomly sampled files (Section II-A).

Membership: If a repository is included in a widely used pretraining dataset, many models have probably seen that repository's code. We do not have direct access to, nor knowledge of, the training datasets for all evaluation models. However, we have partial information about the use of pretraining datasets, such as for open-source models, and closed-source models are likely to use them as well. We therefore assess *membership* via whether a benchmark's repositories are present in `TheStack` [28], a dataset of permissibly licensed source code in 358 programming languages intended for training and fine-tuning code-based models. Given its size and popularity, several models report having trained on it, such as `StarCoder` 2 [21]; other closed-source models are likely to also use it. We used the *Am I in the Stack* tool[1] on each benchmark repository, across the several versions of `TheStack`.

Negative Log Likelihood (`NLL`): `NLL` evaluates how closely an input sequence aligns with patterns the model has learned during training in terms of how "natural" the sequence appears to the model. If the model has seen a data point during training, we expect it to have a lower `NLL` on that point compared to unseen data. If the model has encountered a data point *many* times during training, `NLL` is expected to be particularly low (i.e., close to zero) compared to arbitrary code.

To compute `NLL`, we use the reference implementation publicly available on HuggingFace.[2] Calculating the exact `NLL` for lengthy sequences is usually impractical because LLMs are trained on the limited context (moreover, we cannot fit an entire sequence in memory). Therefore, we split lengthy solution files into overlapping chunks, processed them individually, and combined them using a striding technique. We use strides of 512 tokens when a sequence does not fit into the model's context window.

N-gram accuracy: N-gram accuracy measures the extent to which a model's output exactly matches a reference sequence at the level of n-grams (i.e., contiguous sequences of n tokens). High n-gram accuracy indicates that the model's output closely resembles the reference text, suggesting memorization. N-gram accuracy of 1.0 indicates the model can produce a sequence verbatim.

We follow prior work [9] and use 5-grams (5-grams strike a balance between compute efficiency and metric accuracy). Since most files cannot fit the context window, we use striding to cover the entire sequence. Following Xu et al. [9], we compute 5-grams from five uniformly distributed starting points per stride. For each starting point, we provide the model with the preceding context, and check whether the predicted string matches ground truth.

[1]https://huggingface.co/spaces/bigcode/in-the-stack
[2]https://huggingface.co/docs/transformers/en/perplexity

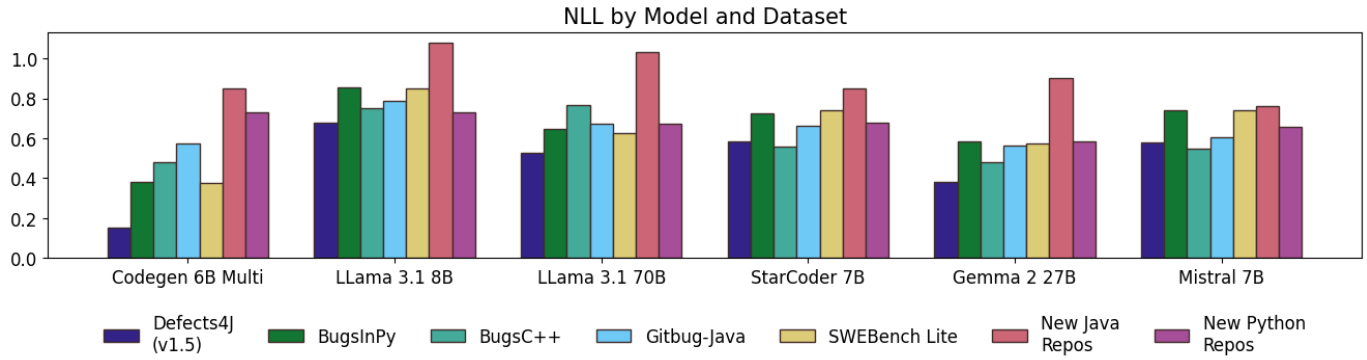


Fig. 3: NLL by model and dataset. NLL is not comparable across models in different families, only across benchmarks within a family. NLL for other models are consistent with the results displayed.

TABLE III: Percentage of repositories in each benchmark leaked in TheStack versions 1.0, 2.0 and 2.1.

Benchmark	v1.0 (%)	v2.0 (%)	v2.1 (%)
GitBug-Java	61.1	42.6	38.9
BugsInPy	94.1	64.7	64.7
BugsC++	60.9	60.9	65.2
Defects4J	80.0	80.0	80.0
SWEBench-Lite	83.3	91.7	83.3

III. Results

This section presents results assessing possible leakage of bug benchmarks in base models, using the metrics described in Section II-C: membership in TheStack (Section III-A), Negative Log Likelihood (Section III-B), and 5-gram accuracy (Section III-C). We also perform a regression analysis of model characteristics, NLL, and 5-gram accuracy (Section III-D) to better understand characteristics of models that influence data leakage. (Note that we subsequently discuss implications in Section IV, and limitations and threats to the validity of our experiments in Section V.)

A. Membership

Table III shows benchmark membership in three versions of TheStack.[3] The table excludes our new Java and Python data from 2024, as TheStack only includes data to 2023. Of all repositories, the new GitBug-Java benchmark has the lowest membership. TheStack contains high proportions of Defects4J and SWEBench-Lite.

While membership does not necessarily mean a model trained on TheStack has seen a specific fixed file (e.g., if a bug-fixing patch was applied after the dataset's cut-off date), the model may still be familiar with a project's source code. This familiarity could lead to higher-quality patches or results. This is not inherently problematic but is a critical factor to consider when assessing the model's potential for generalization.

B. Negative Log Likelihood

Figure 3 shows NLL values for families of open-source models, allowing us to examine trends in familiarity across benchmarks.Note that Negative Log Likelihood (NLL) depends on tokenization and architecture. This means we can only directly compare NLL values within model families.

Figure 3 shows that Defects4J consistently has the lowest NLL across all models. This strongly suggests potential data leakage. This is particularly evident with codegen-multi, which has very low NLL (0.15) for Defects4J. This matches our observations in Figure 1 and suggests that codegen-multi has memorized the Defects4J solutions. We observe comparably low NLL (0.38) on the Gemma 2 27B model for Defects4J relative to other benchmarks and repositories.

Interestingly, codegen-multi 6B exhibits low NLL (0.38) on SWEBench-Lite compared to other benchmarks and new data, despite being the oldest model in our evaluation, trained on older data, and the fact that SWEBench-Lite was published recently. This is because the projects in the benchmark existed prior to benchmark publication, as we also saw in the membership analysis (Table III). Moreover, although SWEBench-Lite is a new benchmark, the bug fixes date as early as 2017.

For all other models, the NLL values are fairly consistent across non-Defects4J benchmarks. As expected, the new repositories we collected exhibit higher NLL compared to Defects4J, BugsCpp, BugsInPy, and SWEBench-Lite. Evaluation benchmarks are derived from prominent projects and may have been seen at a pretraining time multiple times, unlike our new repositories, which likely were not. Note, however, a potential confound, which is that our new repositories may be different in distribution compared to the models' training data, which may contribute to higher NLL.

Figure 4 visualizes how each benchmark compares against other benchmarks and repositories for each model family, demonstrating how much more 'familiar' a particular model is with a benchmark compared to the others. For example, codegen-multi's NLL on Defects4J is $5.63\times$ lower than on new Java repositories, and $3.82\times$ lower than on GitBug-Java. This ratio highlights a significant level of familiarity with Defects4J compared to new repositories. Newer models, particularly LLaMa 3.1, exhibit relatively consistent NLL values across all benchmarks. For example, the NLL ratio between Defects4J and GitBug-Java for LLaMa 3.1 70B is only 1.27, indicating that LLaMa 3.1 perceives Defects4J patches as

[3] V1.0 is the initial 3TB of permissively licensed code, 2.0 expands to 15TB of code, and V2.1 eliminates "opt-out" data.

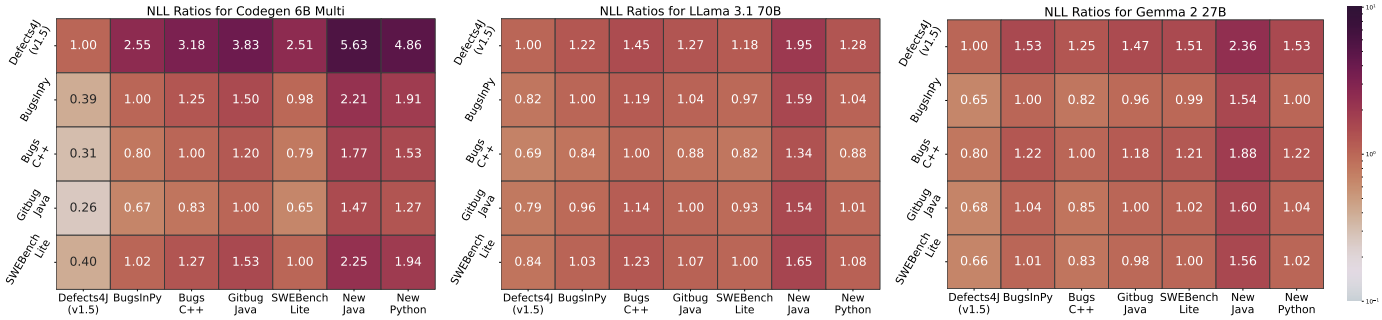


Fig. 4: Heatmap illustrating the relative `NLL` ratios across datasets for the `codegen-multi`, `LLaMa 3.1`, and `Gemma 2`. Each cell represents the ratio of the `NLL` for the dataset in the column to that of the dataset in the row. For example, the `NLL` for new `Java` repos is $5.63\times$ higher than that for `Defects4J`. Darker colors correspond to higher ratios.

only slightly more predictable than `GitBug-Java`. This is expected, as the newer `LLaMa 3.1` family of models was trained on significantly more data and is thus less prone to data memorization. Specifically, the `LLaMa 3.1` family was trained on $30\times$ more tokens than `codegen-multi`.

C. 5-Gram Accuracy

Figure 5 shows `5-gram accuracy` results, a complementary assessment of potential model memorization (full tables, including other models are in the appendix). Note that, due to differences in model vocabularies and tokenization, interpretation of `5-gram accuracy` can differ across models.

`Defects4J` consistently exhibits the highest `5-gram accuracy` across all model families. Conversely, `5-gram accuracy` on `GitBug-Java` is relatively similar to that of new repositories for all models, which aligns with the expectation that most repositories in the `GitBug-Java` benchmark were not included in pretraining data (as detailed in Table I). For example, both `codegen-multi` and `Gemma 2` show significantly higher 5-gram match differences between `Defects4J` and new repositories (i.e., 34 percentage points and 14 percentage points, respectively). Moreover, `codegen-multi` achieves 82% `5-gram accuracy` on `Defects4J`, strongly suggesting that it has likely memorized much of the benchmark's solutions.

As expected, new repositories generally exhibit lower `5-gram accuracy` across all models, with averages of 47% and 48% for `Java` and `Python`, respectively. These values happen likely due to the presence of common coding patterns [29]. Conversely, and in line with our `NLL` findings, `codegen-multi` shows a notably high `5-gram accuracy` on `SWEBench-Lite`, even though it is a recently published dataset, as it incorporates older data.

When it comes to `LLaMa 3.1` family, both `LLaMa 3.1` 70B and 8B exhibit consistent `5-gram accuracy` across benchmarks, which could suggest that these models are less prone to memorization due to their exposure to substantially larger datasets during pretraining. However, it is important to notice that `5-gram accuracy` alone may not conclusively prove an absence of memorization (contrary to high n-gram accuracy which reliably indicates strong pattern retention). For example, Figure 6b shows an example where `LLaMa 3.1` 70B is prompted to predict a patch from `BugsInPy` (`fastapi`:Bug 12). On this file, one of the expected 5-grams is "`__init__(self,`" , and here `LLaMa 3.1` predicts "`__init__(\nself,`" instead. This causes the 5-grams not to match, even though the content is the same. Therefore, it is crucial to evaluate models by considering `NLL`, `5-gram accuracy`, and membership as whole, as no single metric provides a complete picture of data leakage.

D. How Do Model Characteristics Influence Risk of Leakage?

To gain deeper insights, and using the data collected during the study, we estimate regression models for the average values of `NLL` and `5-gram accuracy`. The regressions allow us to explore relationships that metrics alone cannot reveal. Using regression analysis, we can identify factors such as training parameters and budget as significant influences on these metrics. Therefore, we use a simple mixed-effects linear model to predict the average `NLL` and `5-gram accuracy`. As predictors, we include the models' number of *parameters* and number of tokens used during pretraining (i.e., *training budget*), which serves as a proxy for the unique token count. We acknowledge that reporting the exact number of unique tokens would be a more precise metric, but such data is often unavailable. We also account for potential variability caused by differences in the datasets and tokenizers by including *dataset* and *tokenizer type* as random effect. Table IV shows results.

In a linear regression, the intercept represents the predicted value of the dependent variable (averages of `NLL` and `5-gram accuracy`) when all predictors are at their reference values. To make the interpretation easier, we centered the predictors around `codegen-multi` values. Therefore, the reference value of *parameters* is 6B and the reference value of *training budget* is 1T tokens. The regression coefficients represent the change in the dependent variable for a one-unit increase in the predictor, holding other predictors constant.

For `NLL`, the predicted average is 0.744 when at reference level, i.e., parameters = 6B and training budget = 1T. For every 1B increase in number of parameters (above 6B), the predicted `NLL` average decreases by 0.002 units (while holding training budget constant, at 1T). Similarly, for every 1T tokens increase in training budget, predicted `NLL` decreases by 0.014 units. For

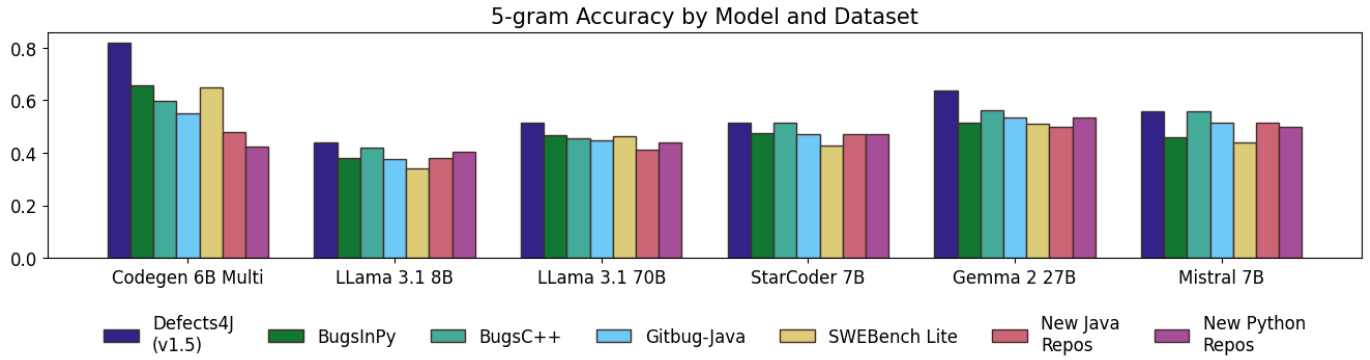


Fig. 5: `5-gram accuracy` by model and dataset. Due to space constraints, we selected a sample of the most relevant models. `5-gram accuracy` for other models are consistent with the results displayed.

TABLE IV: Summary of regressions tests for negative log likelihood (`NLL`) and `5-gram accuracy`. We report the coefficient estimates with their standard errors in parentheses.

	NLL	**5-gram**
Intercept	0.744 (0.094) ***	0.465 (0.049) ***
Parameters	-0.002 (0.001) *	0.001 ($4.0e{-4}$) *
Training budget	-0.014 (0.004) **	0.006 ($2.4e{-3}$) *

*Note: *** p-value < 0.001, ** p-value < 0.01, * p-value < 0.05*

`5-gram accuracy`, the predicted average is 0.465 when predictors are at reference level. For every 1B increase in number of parameters, the predicted `5-gram accuracy` increases by 0.001 units. Similarly, a 1-unit increase in the training budget leads to an increase of 0.006 in `5-gram accuracy`. All reported coefficients have statistical significance.

Regression results reveal consistent trends across model families. For example, both `LLaMa 3.1` 8B and 70B were trained on the same data with a training budget of 15T tokens. However, the 70B model is approximately ten times larger than the 8B model, leading to an overall increase in `5-gram accuracy` across all benchmarks (as shown in Figure 5) and decrease in `NLL`. Similarly, within the `Gemma 2` family, the 2B model was trained on 3T tokens, while the 27B was trained on 12T. Here, we observe the same trend: average `5-gram accuracy` increases in the 27B model and `NLL` decreases.

Regression results also imply that models with more parameters tend to exhibit higher `n-gram accuracy` and, consequently, memorize more. For example, Figure 6b shows that `LLaMa 3.1` 70B, which has the same training budget as `LLaMa 3.1` 8B, accurately predicts strings, class names, and if-statements in contexts where these predictions might not be immediately apparent, which may indicate memorization.

IV. Discussion

Our evaluation provides compelling evidence that data leakage is an especially significant issue for `Defects4J` (V1.5). This is evident from the lower `NLL` values and higher `5-gram accuracy`. In particular, `codegen-multi` achieves 82% `5-gram accuracy` on `Defects4J`, while both `CodeLlama 7B` and `Gemma-2 27B` attain 64%. Moreover, given that `Defects4J` is incorporated into the widely-used pretraining dataset (`TheStack`), with 80% membership, eliminating this leakage in future models is likely nearly impossible. We also observe similarly low `NLL` values in `SWEBench-Lite` for `codegen-multi`.

Newer benchmarks `BugsInPy` and `BugsCpp` exhibit lower leakage risk in almost all models. A smaller percentage of their repositories are indexed in the latest version of `TheStack`. While `BugsInPy` and `BugsCpp` exhibit slightly lower `NLL` compared to new repositories, their `5-gram accuracy` and `NLL` values are not significantly different from newer, more-likely-unseen benchmarks like `GitBug-Java`. This suggests that, for now, researchers can use these benchmarks with a relatively low risk of data leakage. However, as pretraining datasets continue to evolve, we recommend that researchers regularly assess `TheStack` membership, `5-gram accuracy` and `NLL` values of these benchmarks, especially compared to more recent data, to monitor and mitigate potential data contamination.

We also observe that the `LLaMa 3.1` family seems to exhibit lower memorization of benchmark solutions. Nonetheless, we still observed cases where `LLaMa 3.1` 70B outputs solution files despite little context (e.g., Figure 6).

We suggest researchers consider supplementing their evaluations with more recent benchmarks such as `GitBug-Java`. Benchmarks like `GitBug-Java`, which focus on recent bugs and patches, are less likely to have been included in pretraining datasets compared to established benchmarks. Leveraging these newer benchmarks can provide more reliable evaluations for assessing model's capabilities.

V. Limitations and Threats

Data Collection: Despite efforts to filter out older `GitHub` repositories when collecting new data, we anecdotally observed instances where files appeared to be adaptations of existing files (e.g., from 2018) Our filtering process may not have perfectly excluded legacy code. We moreover cannot guarantee that the new repositories are identically distributed compared to those in the benchmarks. To mitigate this issue, given that the repositories in the benchmarks tend to be highly recognizable projects, we applied a >100-star filter to argue that the selected projects are of comparable quality.

```
                * * omitted for brevity * *
    if scheme.lower() != "bearer":
        if self.auto_error:
            raise HTTPException(
              status_code=HTTP_403_FORBIDDEN,
              detail="Invalid authentication
                      credentials scheme")
        else:
            return None
    return HTTPAuthorizationCredentials(scheme=scheme,
credentials=credentials) self.model.parse(credentials)

class HTTPDigest(HTTPBase) BearerModel(BaseModel)::
    def __init__(self, *, bearerFormat: str = None,
                * * omitted for brevity * *
```

patch

- Prompt input to `LLaMa 3.1 8B`.
- Matching Ground Truth & Predicted by `LLaMa 3.1 8B`.
- Extra code generated.
- Ground truth not generated.

(a) Example of generation by `LLaMa 3.1 8B`.

```
                * * omitted for brevity * *
    if scheme.lower() != "bearer":
        if self.auto_error:
            raise HTTPException(
            status_code=HTTP_403_FORBIDDEN,\n
            detail="Invalid authentication credentials"\n
        )
        else:
            return None
    return HTTPAuthorizationCredentials(scheme=scheme,
                                credentials=credentials)

class HTTPDigest(HTTPBase):
    def __init__(\nself,\n*,\n qop: str = None \n,
                * * omitted for brevity * *
```

patch

- Prompt input to `LLaMa 3.1 70B`.
- Matching Ground Truth & Predicted by `LLaMa 3.1 70B`.
- Extra code generated.
- Ground truth not generated.

(b) Example of generation by `LLaMa 3.1 70B`.

Fig. 6: Example patch of a bug from BugsInPy (`fastapi`:Bug 12). We prompt each model with the 30 lines prior to the patch. We highlight the extra code generated not in the ground truth and ground truth code not generated by the model. The remaining lines of the examples are omitted for brevity.

Train + Test Splits: For benchmarks like `Defects4J` (V1.5) and `BugsInPy`, the patch files and buggy files are very similar (typically a patch involves only changing a small number of lines). LLMs may have only seen the train split of these benchmarks at pretraining time. This would result in high `5-gram accuracy` and low `NLL` on patch files, even if only buggy files were leaked. Nonetheless, we observed multiple cases where models output patched files verbatim. We mitigate this by looking at trends in NLL and `5-gram accuracy` rather than absolute numbers in our analysis.

Forgetting: The findings presented in this paper primarily address leakage in the context of base models. Empirical results show that models can "forget" portions of their pretraining data during fine-tuning [30]. That said, while full-scale model fine-tuning may reduce leakage risks, more recent fine-tuning strategies—such as the addition of adapter layers—often "freeze" pretrained weights. This practice can inadvertently increase the likelihood of data leakage. Furthermore, larger models exhibit a stronger tendency to memorize training data [31], suggesting that while fine-tuning may mitigate leakage for smaller models, it is more likely to exacerbate leakage concerns in larger models. We leave empirical assessment of these risks to future work.

VI. Related Work

Large Language Models (LLM): LLMs have shown promise across a wide range of natural language [32], [33] and code generation tasks [34], [35], [36]. Current state of the art LLMs include open source models such as `codegen-multi`, `CodeLLaMa`, `LLaMa 3.1`, `Gemma 2` and and closed-source models such as `GPT-4o`, `Claude 3.5`, and `Gemini 1.5`. LLMs have shown state of the art performance in APR, suggesting patches to buggy code segments, detecting vulnerabilities and helping pinpoint buggy lines in a piece of code [25], [26], [37].

Despite the widespread success of LLMs in APR tasks, there are still significant concerns regarding data leakage. This is prevalent for LLMs trained on large, publicly available code repositories, as these models might "remember" solutions from benchmark datasets, resulting in inflated performance.

Measuring Data Leakage: This task is still an open challenge. Three common ways to measure data leakage are perplexity or `NLL`, `n-gram accuracy`, and model performance. `NLL` based approaches [10], [9] quantify leakage by comparing loss across benchmarks. `n-gram accuracy` approaches [9], [38], prompt a model with context and measure the overlap of generated code with the ground truth. Finally, performance approaches [39], [40] measure the difference in performance between a benchmark and a transformed version of the benchmark. While we build on established techniques from the literature, our work is the first to apply these metrics to bug benchmarks widely recognized in the software engineering community. Additionally, we conduct a comprehensive study of both recent models, such as the `LLaMa 3.1` family, and earlier models like `codegen-multi`.

VII. Conclusion

In this paper, we measure data leakage risks across multiple widely used bug benchmarks and state-of-the-art open-source models. Older benchmarks, especially `Defects4J`, exhibit higher memorization signals across all models, particularly in `codegen-multi` 6B. Newer datasets such as `GitBug-Java`, `BugsInPy`, `BugsCpp` show lower leakage risk, with similar `NLL` and `5-gram accuracy` to new 2024 repositories. We also find that newer models with higher training budgets display

significantly lower risks of data leakage. Our findings suggest that researchers should consider both the models they use in evaluation and pair older benchmarks, such as `Defects4J`, with more recent benchmarks, such as `GitBug-Java`, to improve model evaluation and address potential leakage issues

VIII. Acknowledgments

The authors would like to thank Luke Dramko for providing code that we used for deduplicating 2024 repositories. This work was supported by Fundação para a Ciência e Tecnologia (FCT) through the CMU Portugal Dual PhD program: Daniel Ramos (SFRH/BD/150688/2020), Claudia Mamede (PRT/BD/155043/2022), Paulo Canelas (SFRH/BD/151469/2021), Catarina Gamboa (PRT/BD/154254/2021).

References

[1] M. Chen *et al.*, "Evaluating large language models trained on code," *CoRR*, vol. abs/2107.03374, 2021. [Online]. Available: https://arxiv.org/abs/2107.03374

[2] W. E. Wong *et al.*, "A survey on software fault localization," *IEEE Transactions on Software Engineering*, vol. 42, no. 8, pp. 707–740, 2016.

[3] C. Le Goues, M. Pradel, and A. Roychoudhury, "Automated Program Repair," *Communications of the ACM*, vol. 62, no. 12, pp. 56–65, 2019.

[4] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: a database of existing faults to enable controlled testing studies for java programs," in *International Symposium on Software Testing and Analysis*, 2014, pp. 437–440.

[5] R. Widyasari *et al.*, "Bugsinpy: a database of existing bugs in python programs to enable controlled testing and debugging studies," in *Foundations of Software Engineering*, 2020.

[6] C. E. Jimenez *et al.*, "SWE-bench: Can language models resolve real-world github issues?" in *International Conference on Learning Representations*, 2024. [Online]. Available: https://openreview.net/forum?id=VTF8yNQM66

[7] J. Cao, W. Zhang, and S. Cheung, "Concerned with data contamination? assessing countermeasures in code language model," *CoRR*, vol. abs/2403.16898, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2403.16898

[8] S. Kapoor and A. Narayanan, "Leakage and the reproducibility crisis in ml-based science," *CoRR*, vol. abs/2207.07048, 2022. [Online]. Available: https://doi.org/10.48550/arXiv.2207.07048

[9] R. Xu *et al.*, "Benchmarking benchmark leakage in large language models," *CoRR*, vol. abs/2404.18824, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2404.18824

[10] Y. Li, "Estimating contamination via perplexity: Quantifying memorisation in language model evaluation," *CoRR*, vol. abs/2309.10677, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2309.10677

[11] G. An *et al.*, "Bugsc++: A highly usable real world defect benchmark for c/c++," in *International Conference on Automated Software Engineering*. IEEE Press, 2024, p. 2034–2037. [Online]. Available: https://doi.org/10.1109/ASE56229.2023.00208

[12] A. Silva, N. Saavedra, and M. Monperrus, "Gitbug-java: A reproducible benchmark of recent java bugs," in *International Conference on Mining Software Repositories*, ser. MSR '24. ACM, 2024, p. 118–122.

[13] C. S. Xia *et al.*, "Agentless: Demystifying llm-based software engineering agents," *CoRR*, vol. abs/2407.01489, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2407.01489

[14] X. Wang *et al.*, "Openhands: An open platform for ai software developers as generalist agents," 2024. [Online]. Available: https://arxiv.org/abs/2407.16741

[15] R. Aleithan *et al.*, "Swe-bench+: Enhanced coding benchmark for llms," 2024. [Online]. Available: https://arxiv.org/abs/2410.06992

[16] A. Broder, "On the resemblance and containment of documents," in *Compression and Complexity of SEQUENCES 1997 (Cat. No.97TB100171)*, 1997, pp. 21–29.

[17] A. Gionis, P. Indyk, and R. Motwani, "Similarity search in high dimensions via hashing," in *International Conference on Very Large Data Bases*, ser. VLDB '99, 1999, p. 518–529.

[18] E. Nijkamp *et al.*, "Codegen: An open large language model for code with multi-turn program synthesis," in *International Conference on Learning Representations*. OpenReview.net, 2023.

[19] B. Rozière *et al.*, "Code llama: Open foundation models for code," 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2308.12950

[20] H. Touvron *et al.*, "Llama: Open and efficient foundation language models," 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2302.13971

[21] A. Lozhkov *et al.*, "Starcoder 2 and the stack v2: The next generation," 2024.

[22] G. Team *et al.*, "Gemma 2: Improving open language models at a practical size," 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2408.00118

[23] C. Team *et al.*, "Codegemma: Open code models based on gemma," 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2406.11409

[24] A. Q. Jiang *et al.*, "Mistral 7b," 2023. [Online]. Available: https://arxiv.org/abs/2310.06825

[25] A. Z. H. Yang, C. Le Goues, R. Martins, and V. J. Hellendoorn, "Large language models for test-free fault localization," in *International Conference on Software Engineering*. ACM, 2024, pp. 17:1–17:12.

[26] A. Silva, S. Fang, and M. Monperrus, "Repairllama: Efficient representations and fine-tuned adapters for program repair," *CoRR*, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2312.15698

[27] X. Zhou, T. Zhang, and D. Lo, "Large language model for vulnerability detection: Emerging results and future directions," in *International Conference on Software Engineering: New Ideas and Emerging Results*. ACM, 2024, pp. 47–51.

[28] D. Kocetkov *et al.*, "The stack: 3 TB of permissively licensed source code," *Trans. Mach. Learn. Res.*, 2023. [Online]. Available: https://openreview.net/forum?id=pxpbTdUEpD

[29] M. Gabel and Z. Su, "A study of the uniqueness of source code," in *International Symposium on Foundations of Software Engineering*, G. Roman and A. van der Hoek, Eds. ACM, 2010, pp. 147–156.

[30] Y. Luo *et al.*, "An empirical study of catastrophic forgetting in large language models during continual fine-tuning," *CoRR*, vol. abs/2308.08747, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2308.08747

[31] K. Tirumala, A. H. Markosyan, L. Zettlemoyer, and A. Aghajanyan, "Memorization without overfitting: Analyzing the training dynamics of large language models," in *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems*, 2022.

[32] P. Rajpurkar, J. Zhang, K. Lopyrev, and P. Liang, "Squad: 100, 000+ questions for machine comprehension of text," in *Conference on Empirical Methods in Natural Language Processing*, 2016, pp. 2383–2392. [Online]. Available: https://doi.org/10.18653/v1/d16-1264

[33] D. Hendrycks *et al.*, "Measuring massive multitask language understanding," in *International Conference on Learning Representations1*. OpenReview.net, 2021. [Online]. Available: https://openreview.net/forum?id=d7KBjmI3GmQ

[34] M. Chen *et al.*, "Evaluating large language models trained on code," 2021. [Online]. Available: https://arxiv.org/abs/2107.03374

[35] J. Austin *et al.*, "Program synthesis with large language models," *CoRR*, 2021. [Online]. Available: https://arxiv.org/abs/2108.07732

[36] D. Hendrycks *et al.*, "Measuring coding challenge competence with APPS," in *Neural Information Processing Systems Track on Datasets and Benchmarks*, J. Vanschoren and S. Yeung, Eds., 2021.

[37] C. S. Xia and L. Zhang, "Keep the conversation going: Fixing 162 out of 337 bugs for $0.42 each using chatgpt," *CoRR*, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2304.00385

[38] W. Shi *et al.*, "Detecting pretraining data from large language models," in *International Conference on Learning Representations*. OpenReview.net, 2024. [Online]. Available: https://openreview.net/forum?id=zWqr3MQuNs

[39] N. Jain *et al.*, "Livecodebench: Holistic and contamination free evaluation of large language models for code," *CoRR*, vol. abs/2403.07974, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2403.07974

[40] C. Zuo, Z. Lin, and Y. Zhang, "Why does your data leak? uncovering the data leakage in cloud from mobile apps," in *Symposium on Security and Privacy*, 2019, pp. 1296–1310.

APPENDIX A

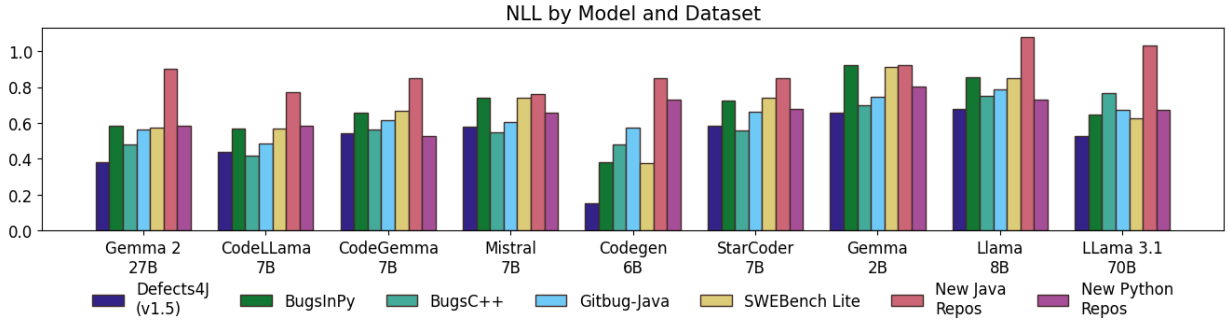


Fig. 7: NLL results for all the models and benchmarks.

TABLE V: Average NLL across benchmarks and models

	Defects4J (V1.5)	BugsInPy	New Java Repos	GitBug Java	BugsCpp	SWE Bench Lite	New Python Repos
Salesforce/codegen-6B-multi	0.15	0.38	0.85	0.58	0.48	0.38	0.73
meta-llama/CodeLlama-7b-hf	0.44	0.57	0.77	0.48	0.42	0.57	0.58
LlaMa-3.1-8B	0.68	0.85	1.08	0.79	0.75	0.85	0.73
starcoder2-7b	0.58	0.72	0.85	0.66	0.56	0.74	0.68
codegemma-7b	0.54	0.66	0.85	0.62	0.57	0.67	0.53
gemma-2-2b	0.66	0.92	0.92	0.74	0.70	0.91	0.80
Mistral-7B-v0.3	0.58	0.74	0.76	0.61	0.55	0.74	0.66
Gemma 2 27b	0.38	0.58	0.90	0.56	0.48	0.58	0.59
LlaMa 70B	0.53	0.65	1.03	0.67	0.77	0.63	0.68

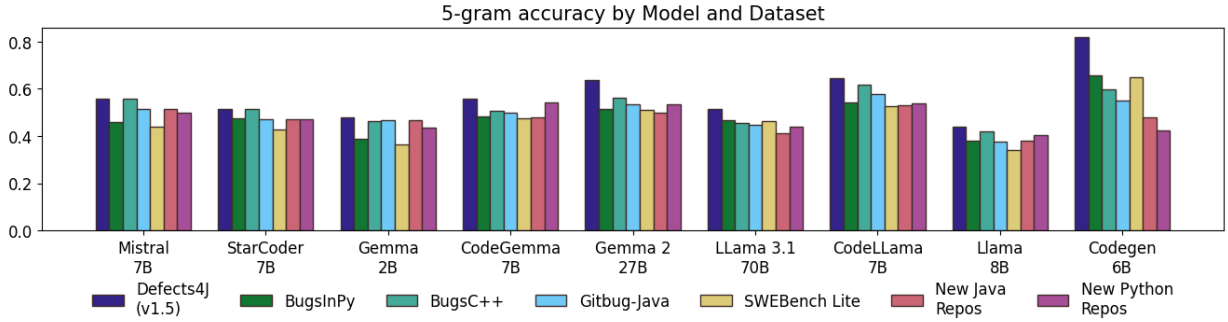
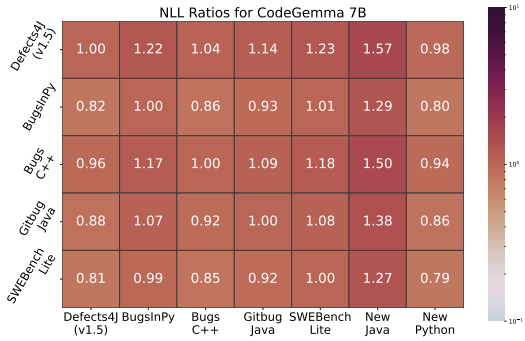


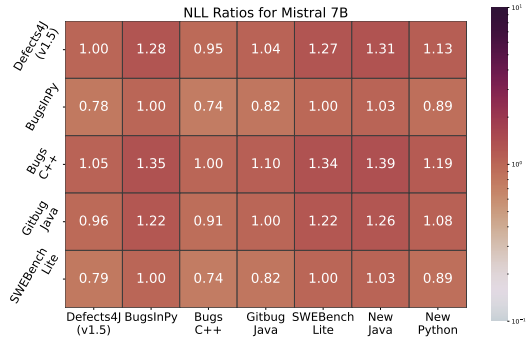
Fig. 8: 5-gram results for all the models and benchmarks.

TABLE VI: Average 5-gram accuracy across multiple benchmarks

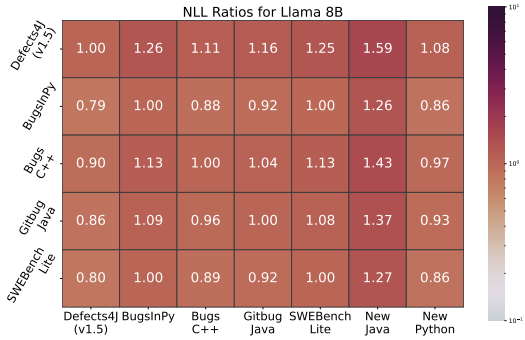
	Defects4J (V1.5)	BugsInPy	New Java Repos	GitBug Java	BugsCpp	SWEBench Lite	New Python Repos
Salesforce/codegen-6B-multi	0.82	0.66	0.48	0.55	0.60	0.65	0.42
meta-llama/CodeLlama-7b-hf	0.64	0.54	0.53	0.58	0.62	0.53	0.54
LlaMa-3.1-8B	0.44	0.38	0.38	0.38	0.42	0.34	0.41
starcoder2-7b	0.51	0.47	0.47	0.47	0.51	0.43	0.47
codegemma-7b	0.56	0.48	0.48	0.50	0.51	0.48	0.54
gemma-2-2b	0.48	0.39	0.47	0.47	0.46	0.37	0.44
Mistral-7B-v0.3	0.56	0.46	0.51	0.52	0.56	0.44	0.50
Gemma 2 27b	0.64	0.52	0.50	0.53	0.56	0.51	0.53
LlaMa 70B	0.51	0.47	0.41	0.45	0.45	0.46	0.44



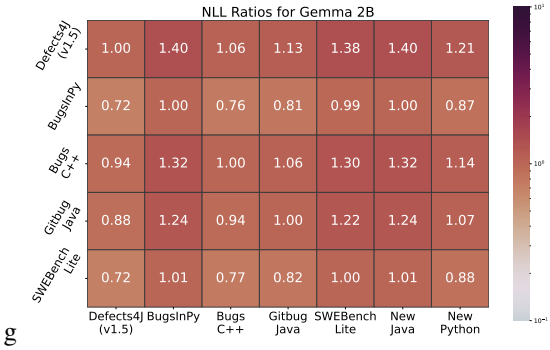
(a) NLL ratios for CodeGemma 7B.



(b) NLL ratios for Mistral 7B.

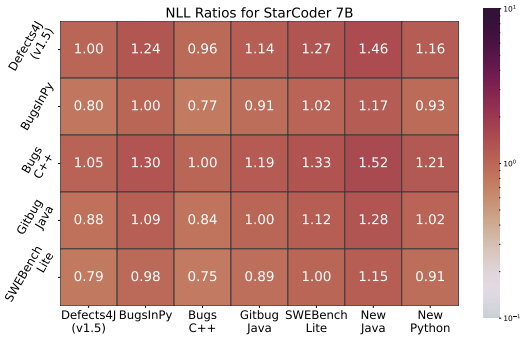


(c) NLL ratios for LlaMa 3.1 8B.

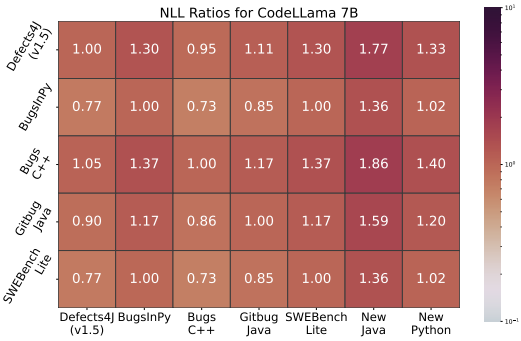


g

(d) NLL ratios for Gemma 2B.

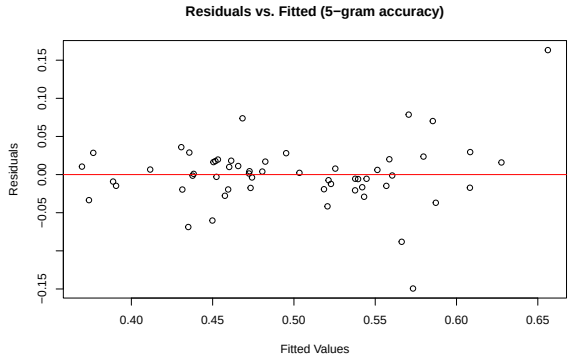


(e) NLL ratios for StarCoder 7B.

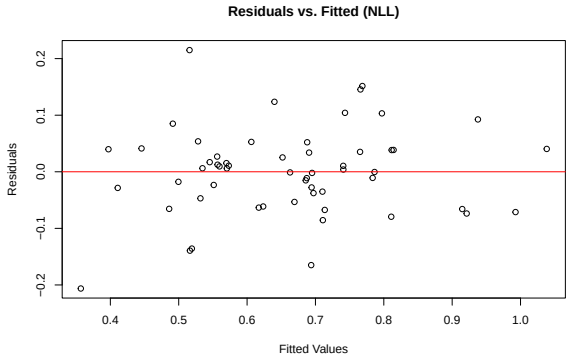


(f) NLL ratios for CodeLLaMa 7B.

Fig. 9: Heatmaps for `NLL` ratios



(a) Residual vs. Fitted Plot for average `5-gram accuracy`.



(b) Residual vs. Fitted Plot for average `NLL`.

Fig. 10: Residuals for the linear regressions.