

Are Large Language Models Memorizing Bug Benchmarks?



**Daniel
Ramos**



**Claudia
Mamede**



**Kush
Jain**



**Paulo
Canelas**



**Catarina
Gamboa**



**Claire
Le Goues**

Large Language Models are everywhere yet there is still a big gap between benchmark performance and reality

Large Language Models are everywhere yet there is still a big gap between benchmark performance and reality

Codegen-6B for bug repair

Prompt

```
1 int start = 0;
2 // get a good guess on the size of the result buffer so
3 // it doesnt have to double if it goes over a bit
4 int increase = 0;
5 // count the replacement text elements that are larger
6 // than their corresponding text being replaced
```

Defects4J, Bug 39 (in Lang project)

Large Language Models are everywhere yet there is still a big gap between benchmark performance and reality

Codegen-6B for bug repair

Prompt

```
1 int start = 0;
2 // get a good guess on the size of the result buffer so
3 // it doesnt have to double if it goes over a bit
4 int increase = 0;
5 // count the replacement text elements that are larger
6 // than their corresponding text being replaced
```

Generated

```
8 for (int i = 0; i < searchList.length; i++) {
9     if (searchList[i] = null || replacementList[i] = null) {
10         continue;
11     }
12     int greater = replacementList[i].length() - searchList[i].length();
13     if (greater > 0) {
14         increase += 3 * greater; // assume 3 matches
15     }
16 }
17 // have upper-bound at 20% increase, then let Java take ...
```

Defects4J, Bug 39 (in Lang project)

Large Language Models are everywhere yet there is still a big gap between benchmark performance and reality

Codegen-6B for bug repair

Prompt

```
1 int start = 0;
2 // get a good guess on the size of the result buffer so
3 // it doesnt have to double if it goes over a bit
4 int increase = 0;
5 // count the replacement text elements that are larger
6 // than their corresponding text being replaced
7
8 for (int i = 0; i < searchList.length; i++) {
9     if (searchList[i] = null || replacementList[i] = null) {
10         continue;
11     }
12     int greater = replacementList[i].length() - searchList[i].length();
13     if (greater > 0) {
14         increase += 3 * greater; // assume 3 matches
15     }
16 }
17 // have upper-bound at 20% increase, then let Java take ...
```

Generated

Original fix file

```
1 int start = 0;
2 // get a good guess on the size of the result buffer so
3 // it doesnt have to double if it goes over a bit
4 int increase = 0;
5 // count the replacement text elements that are larger
6 // than their corresponding text being replaced
7
8 for (int i = 0; i < searchList.length; i++) {
9     if (searchList[i] = null || replacementList[i] = null) {
10         continue;
11     }
12     int greater = replacementList[i].length() - searchList[i].length();
13     if (greater > 0) {
14         increase += 3 * greater; // assume 3 matches
15     }
16 }
17 // have upper-bound at 20% increase, then let Java take ...
```

Defects4J, Bug 39 (in Lang project)

Large Language Models are everywhere yet there is still a big gap between benchmark performance and reality

Codegen-6B for bug repair

Prompt

```
1 int start = 0;
2 // get a good guess on the size of the result buffer so
3 // it doesnt have to double if it goes over a bit
4 int increase = 0;
5 // count the replacement text elements that are larger
6 // than their corresponding text being replaced
7
8 for (int i = 0; i < searchList.length; i++) {
9     if (searchList[i] = null || replacementList[i] = null) {
10         continue;
11     }
12     int greater = replacementList[i].length() - searchList[i].length();
13     if (greater > 0) {
14         increase += 3 * greater; // assume 3 matches
15     }
16 }
17 // have upper-bound at 20% increase, then let Java take ...
```

Generated

Original fix file

```
1 int start = 0;
2 // get a good guess on the size of the result buffer so
3 // it doesnt have to double if it goes over a bit
4 int increase = 0;
5 // count the replacement text elements that are larger
6 // than their corresponding text being replaced
7
8 for (int i = 0; i < searchList.length; i++) {
9     if (searchList[i] = null || replacementList[i] = null) {
10         continue;
11     }
12     int greater = replacementList[i].length() - searchList[i].length();
13     if (greater > 0) {
14         increase += 3 * greater; // assume 3 matches
15     }
16 }
17 // have upper-bound at 20% increase, then let Java take ...
```

Defects4J, Bug 39 (in Lang project)

Large Language Models are everywhere yet there is still a big gap between benchmark performance and reality

Codegen-6B for bug repair

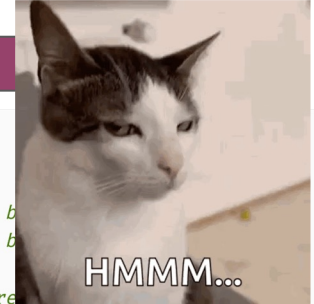
Prompt

```
1 int start = 0;
2 // get a good guess on the size of the result buffer so
3 // it doesnt have to double if it goes over a bit
4 int increase = 0;
5 // count the replacement text elements that are larger
6 // than their corresponding text being replaced
7
8 for (int i = 0; i < searchList.length; i++) {
9     if (searchList[i] = null || replacementList[i] = null) {
10         continue;
11     }
12     int greater = replacementList[i].length() - searchList[i].length();
13     if (greater > 0) {
14         increase += 3 * greater; // assume 3 matches
15     }
16 }
17 // have upper-bound at 20% increase, then let Java take ...
```

Generated

Original fix file

```
1 int start = 0;
2 // get a good guess on the size of the result b
3 // it doesnt have to double if it goes over a b
4 int increase = 0;
5 // count the replacement text elements that are
6 // than their corresponding text being replaced
7
8 for (int i = 0; i < searchList.length; i++) {
9     if (searchList[i] = null || replacementList[i] = null) {
10         continue;
11     }
12     int greater = replacementList[i].length() - searchList[i].length();
13     if (greater > 0) {
14         increase += 3 * greater; // assume 3 matches
15     }
16 }
17 // have upper-bound at 20% increase, then let Java take ...
```



Defects4J, Bug 39 (in Lang project)

Memorization often happens due to data leakage which is hard to detect and quantify 🙄

Memorization often happens due to data leakage which is hard to detect and quantify 🙄



We don't know if the
model was trained
on the test set

Memorization often happens due to data leakage which is hard to detect and quantify 🙄



We don't know if the model was trained on the test set



Model weights are often unavailable

Memorization often happens due to data leakage which is hard to detect and quantify 🙄



We don't know if the model was trained on the test set



Model weights are often unavailable



Hard to define a metric to detect leakage

Memorization often happens due to data leakage which is hard to detect and quantify 🙄

Are large language models memorizing bug benchmarks?

We don't know if the model was trained on the test set

Model weights are often unavailable

Hard to define a metric to detect leakage

We designed an experimental setup to detect memorization across models and bug benchmarks

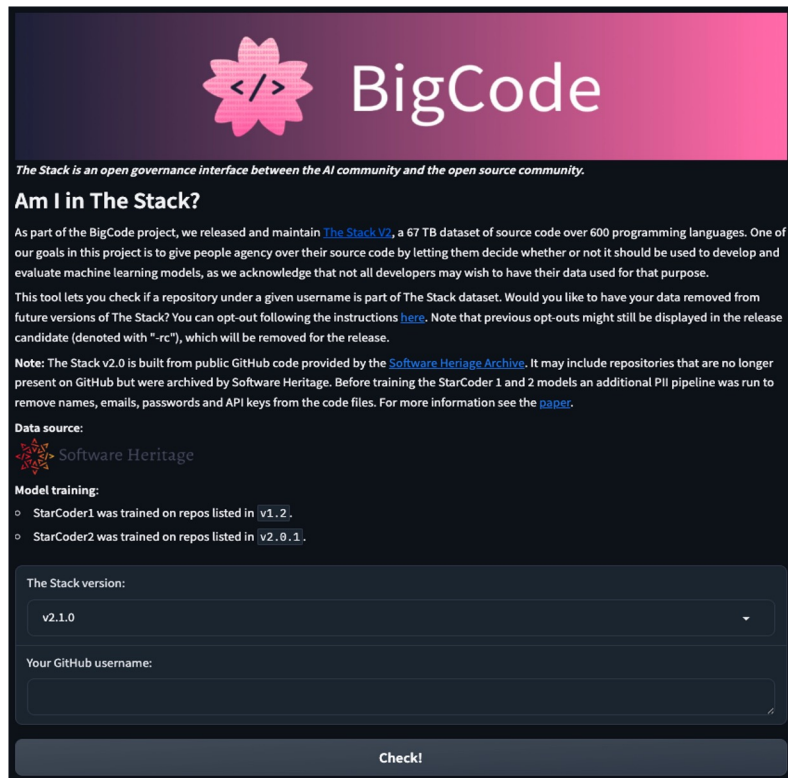
LEAKAGE DETECTION

We designed an experimental setup to detect memorization across models and bug benchmarks

LEAKAGE DETECTION

MEMBERSHIP

~ is my data part of The Stack ?



The screenshot shows the BigCode website interface for the 'Am I in The Stack?' tool. At the top, there is a pink header with the BigCode logo (a pink flower with code symbols) and the text 'BigCode'. Below the header, a dark blue banner contains the text: 'The Stack is an open governance interface between the AI community and the open source community.' The main heading is 'Am I in The Stack?'. The text explains that as part of the BigCode project, they released and maintain 'The Stack v2', a 67 TB dataset of source code over 600 programming languages. It states their goal is to give people agency over their source code by letting them decide whether or not it should be used to develop and evaluate machine learning models. The tool allows users to check if a repository under a given username is part of The Stack dataset and offers an option to opt-out. A note mentions that the Stack v2.0 is built from public GitHub code provided by the Software Heritage Archive and includes a PII pipeline to remove names, emails, passwords, and API keys. The 'Data source:' section features the Software Heritage logo. The 'Model training:' section lists that StarCoder1 was trained on repos listed in v1.2 and StarCoder2 was trained on repos listed in v2.0.1. At the bottom, there is a form with a dropdown menu for 'The Stack version:' set to 'v2.1.0', a text input field for 'Your GitHub username:', and a 'Check!' button.

The Stack is an open governance interface between the AI community and the open source community.


Am I in The Stack?

As part of the BigCode project, we released and maintain [The Stack v2](#), a 67 TB dataset of source code over 600 programming languages. One of our goals in this project is to give people agency over their source code by letting them decide whether or not it should be used to develop and evaluate machine learning models, as we acknowledge that not all developers may wish to have their data used for that purpose.

This tool lets you check if a repository under a given username is part of The Stack dataset. Would you like to have your data removed from future versions of The Stack? You can opt-out following the instructions [here](#). Note that previous opt-outs might still be displayed in the release candidate (denoted with "-rc"), which will be removed for the release.

Note: The Stack v2.0 is built from public GitHub code provided by the [Software Heritage Archive](#). It may include repositories that are no longer present on GitHub but were archived by Software Heritage. Before training the StarCoder 1 and 2 models an additional PII pipeline was run to remove names, emails, passwords and API keys from the code files. For more information see the [paper](#).

Data source:

 Software Heritage

Model training:

- StarCoder1 was trained on repos listed in [v1.2](#).
- StarCoder2 was trained on repos listed in [v2.0.1](#).

The Stack version:

v2.1.0

Your GitHub username:

Check!

We designed an experimental setup to detect memorization across models and bug benchmarks

LEAKAGE DETECTION

MEMBERSHIP

~ is my data part of The Stack ?

NEG. LOG LIKELIHOOD

~ how surprised is the model by my input ?

We designed an experimental setup to detect memorization across models and bug benchmarks

LEAKAGE DETECTION

MEMBERSHIP

~ is my data part of The Stack ?

NEG. LOG LIKELIHOOD

~ how surprised is the model by my input ?

5-GRAM ACCURACY

~ how close is the model output to the ground truth ?

We designed an experimental setup to detect memorization across models and bug benchmarks

LEAKAGE DETECTION

MEMBERSHIP

~ is my data part of The Stack ?

NEG. LOG LIKELIHOOD

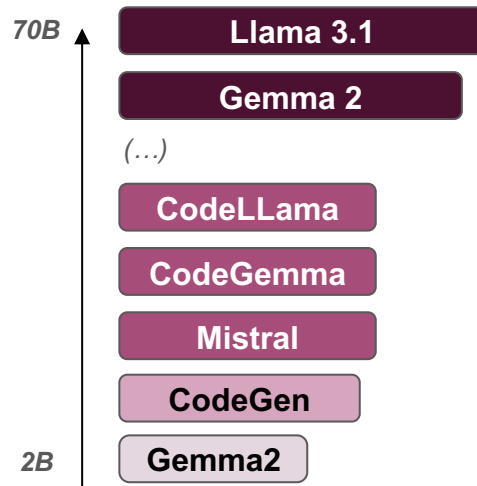
~ how surprised is the model by my input ?

5-GRAM ACCURACY

~ how close is the model output to the ground truth ?

MODEL SELECTION

9 open-source models



We designed an experimental setup to detect memorization across models and bug benchmarks

LEAKAGE DETECTION

MEMBERSHIP

~ is my data part of The Stack ?

NEG. LOG LIKELIHOOD

~ how surprised is the model by my input ?

5-GRAM ACCURACY

~ how close is the model output to the ground truth ?

MODEL SELECTION

9 open-source models

70B

Llama 3.1

Gemma 2

(...)

CodeLLama

CodeGemma

Mistral

CodeGen

2B

Gemma2

BENCHMARKS

5 Bug Benchmarks

BugsC++

GitBug-Java

BugsInPy

Defects4J

Swebench-Lite



New, likely unseen data

Github repos in
Java and Python

Our findings show that...

MEMBERSHIP

All benchmarks were part of The Stack to some degree.

Dataset	Year	Membership (%)		
		v1.0	v2.0	v2.1
Defects4J	2019	80.0	80.0	80.0
BugsInPy	2020	94.1	64.7	64.7
BugsC++	2021	60.9	60.9	65.2
Gitbug-Java	2023	61.1	42.6	38.9
SweBench-Lite	2024	83.3	91.7	83.3

Our findings show that...

MEMBERSHIP

All benchmarks were part of The Stack to some degree.

👎 D4J, SWEBench Lite

Dataset	Year	Membership (%)		
		v1.0	v2.0	v2.1
Defects4J	2019	80.0	80.0	80.0
BugsInPy	2020	94.1	64.7	64.7
BugsC++	2021	60.9	60.9	65.2
Gitbug-Java	2023	61.1	42.6	38.9
SweBench-Lite	2024	83.3	91.7	83.3

Our findings show that...

MEMBERSHIP

All benchmarks were part of The Stack to some degree.

👎 D4J, SWEBench Lite

👍 Gitbug-Java

Dataset	Year	Membership (%)		
		v1.0	v2.0	v2.1
Defects4J	2019	80.0	80.0	80.0
BugsInPy	2020	94.1	64.7	64.7
BugsC++	2021	60.9	60.9	65.2
Gitbug-Java	2023	61.1	42.6	38.9
SweBench-Lite	2024	83.3	91.7	83.3

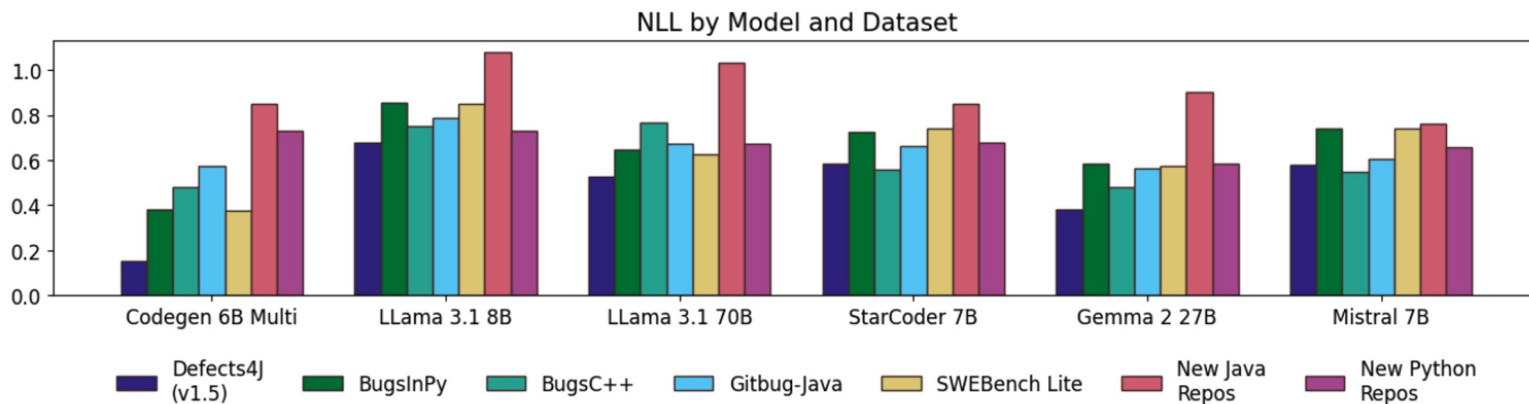
Our findings show that...

MEMBERSHIP

All benchmarks were part of The Stack to some degree.

NEG LOG LIKELIHOOD

Prominent benchmarks elicit lower NLLs across all models.



Our findings show that...

MEMBERSHIP

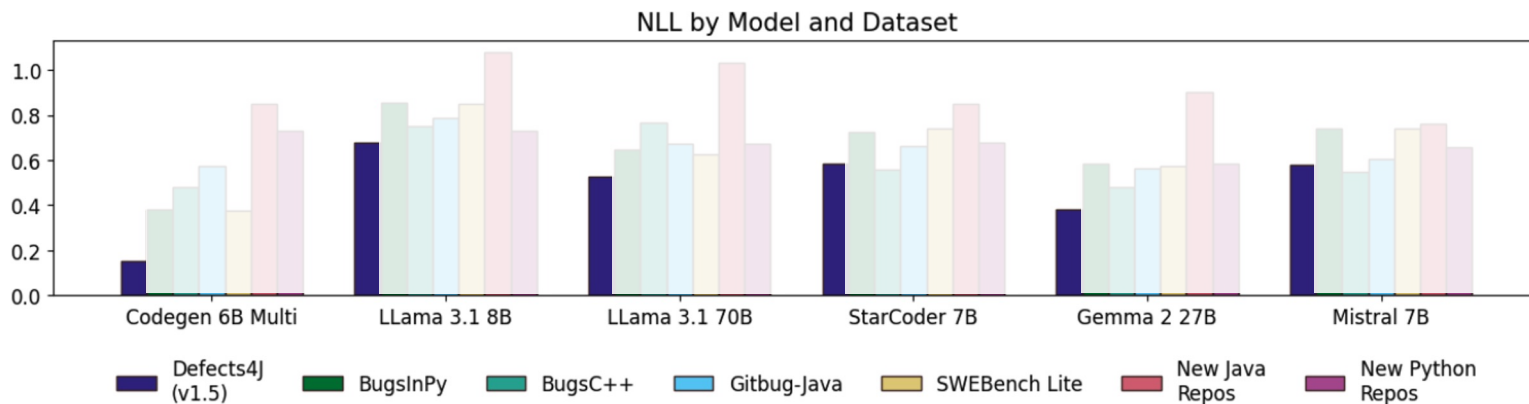
All benchmarks were part of The Stack to some degree.

NEG LOG LIKELIHOOD

Prominent benchmarks elicit lower NLLs across all models.



D4J, SWEBench Lite



Our findings show that...

MEMBERSHIP

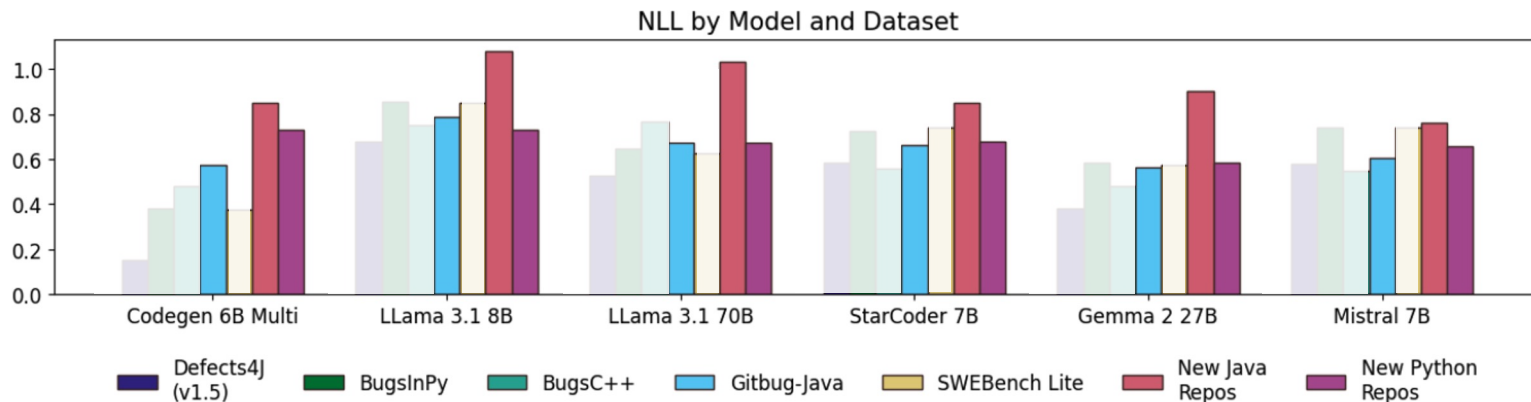
All benchmarks were part of The Stack to some degree.

NEG LOG LIKELIHOOD

Prominent benchmarks elicit lower NLLs across all models.

👎 D4J, SWEBench Lite

👍 New and unseen data, Gitbug-Java



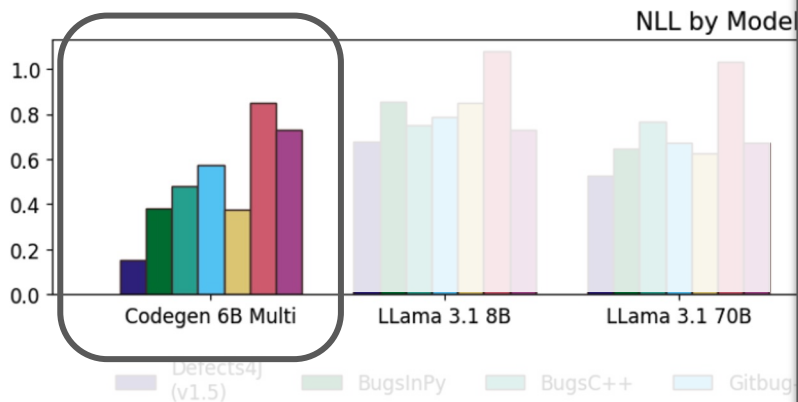
Our findings show that...

MEMBERSHIP

All benchmarks were part

NEG LOG LIKELIHOOD

Prominent bench



NLL Ratios for Codegen 6B Multi

Defects4J (v1.5)	1.00	2.55	3.18	3.83	2.51	5.63	4.86
BugsinPy	0.39	1.00	1.25	1.50	0.98	2.21	1.91
Bugs C++	0.31	0.80	1.00	1.20	0.79	1.77	1.53
Gitbug Java	0.26	0.67	0.83	1.00	0.65	1.47	1.27
SWEBench Lite	0.40	1.02	1.27	1.53	1.00	2.25	1.94

► **Potential memorization signals.**

It is ~5x more familiar with D4J than newer data.

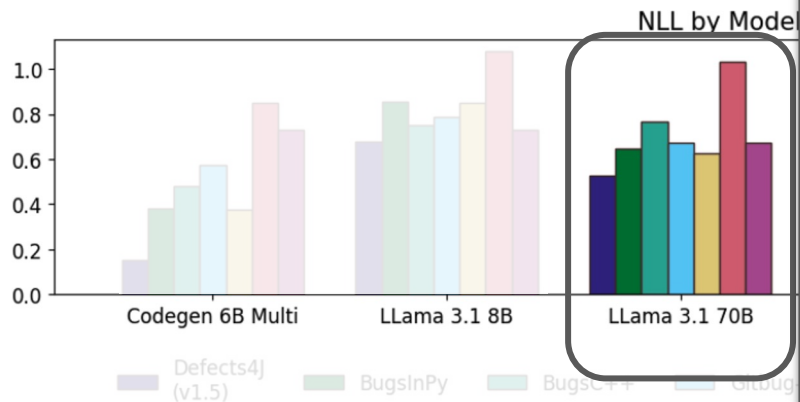
Our findings show that...

MEMBERSHIP

All benchmarks were part

NEG LOG LIKELIHOOD

Prominent bench



NLL Ratios for LLama 3.1 70B

Defects4J (v1.5)	1.00	1.22	1.45	1.27	1.18	1.95	1.28
BugsinPy	0.82	1.00	1.19	1.04	0.97	1.59	1.04
Bugs C++	0.69	0.84	1.00	0.88	0.82	1.34	0.88
Gitbug Java	0.79	0.96	1.14	1.00	0.93	1.54	1.01
SWEBench Lite	0.84	1.03	1.23	1.07	1.00	1.65	1.08
	Defects4J (v1.5)	BugsinPy	Bugs C++	Gitbug Java	SWEBench Lite	New Java	New Python

✨ **More consistent behavior across benchmarks.**

Our findings show that...

MEMBERSHIP

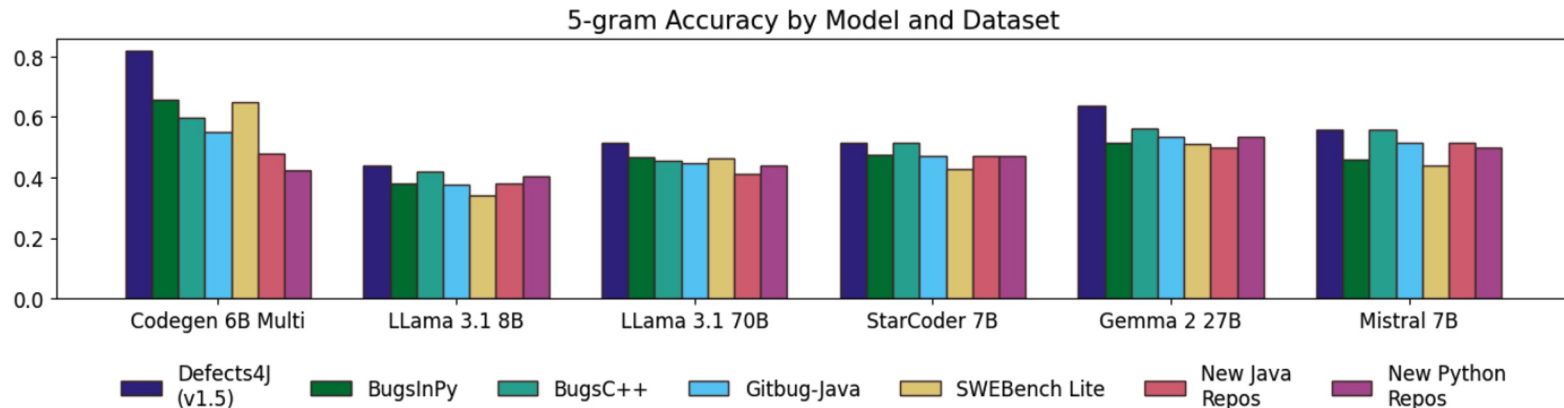
All benchmarks were part of The Stack to some degree.

NEG LOG LIKELIHOOD

Prominent benchmarks elicit lower NLLs across all models.

5-GRAM ACCURACY

Prominent benchmarks elicit higher 5-gram matches across model families.



Our findings show that...

MEMBERSHIP

All benchmarks were part of The Stack to some degree.

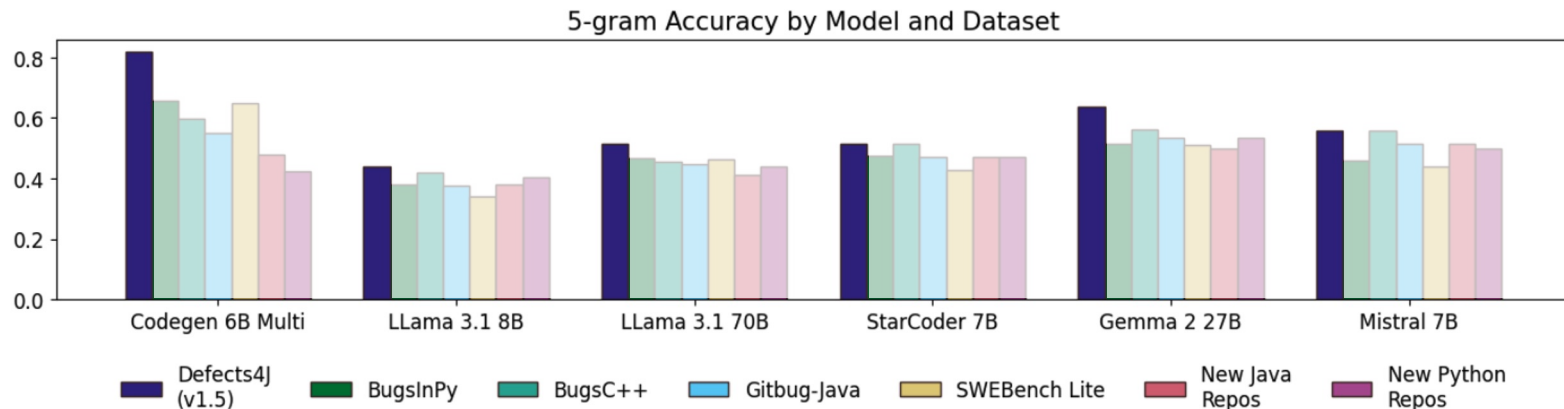
NEG LOG LIKELIHOOD

Prominent benchmarks elicit lower NLLs across all models.

5-GRAM ACCURACY

Prominent benchmarks elicit higher 5-gram matches across model families.

👎 D4J



Our findings show that...

MEMBERSHIP

All benchmarks were part of The Stack to some degree.

NEG LOG LIKELIHOOD

Prominent benchmarks elicit lower NLLs across all models

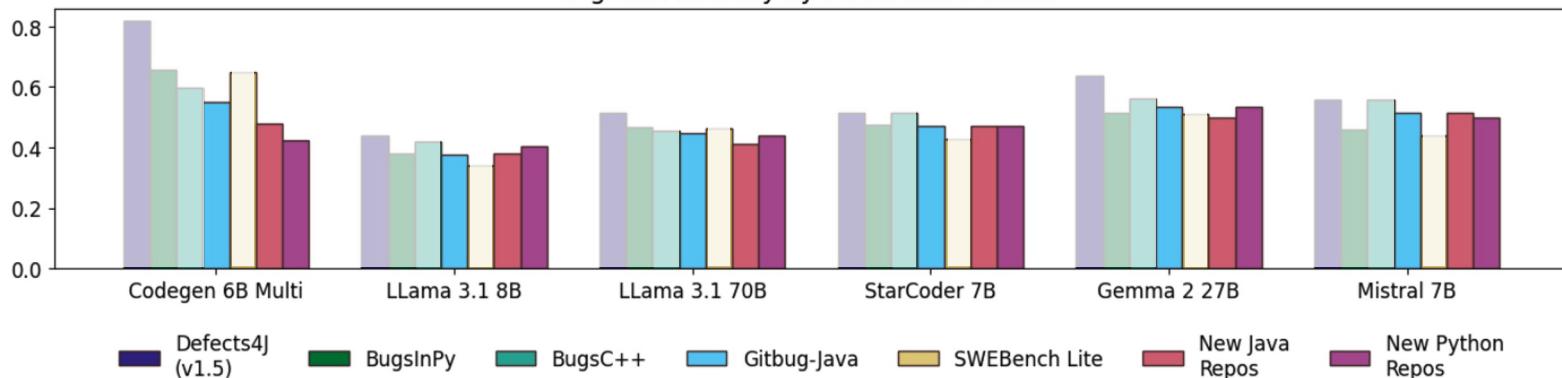
5-GRAM ACCURACY

Prominent benchmarks elicit higher 5-gram matches across model families.

👎 D4J

👍 New and unseen data, Gitbug-Java

5-gram Accuracy by Model and Dataset



Our findings show that...

 **Data leakage is an especially significant issue for Defects4J (V1.5)**

Our findings show that...

 **Data leakage is an especially significant issue for Defects4J (V1.5)**

Smaller models, trained on far less data, seem more prone to memorization

Our findings show that...

 **Data leakage is an especially significant issue for Defects4J (V1.5)**

Smaller models, trained on far less data, seem more prone to memorization

Bigger and more recent models seem to exhibit lower memorization of benchmark solutions

If we are not careful with benchmark selection, we risk reporting inflated model performance due to data leakage

So, we suggest...



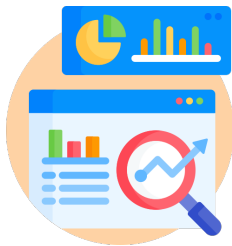
Evaluate on
benchmarks with
new data

If we are not careful with benchmark selection, we risk reporting inflated model performance due to data leakage

So, we suggest...



Evaluate on
benchmarks with
new data



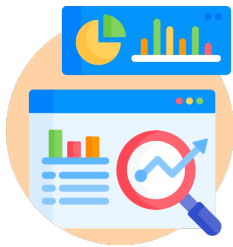
Monitor leakage risk over time
by computing membership,
NLL and 5-gram matches

If we are not careful with benchmark selection, we risk reporting inflated model performance due to data leakage

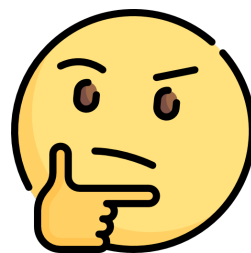
So, we suggest...



Evaluate on benchmarks with new data



Monitor leakage risk over time by computing membership, NLL and 5-gram matches



Carefully interpret benchmarks numbers

Memorization often happens due to data leakage which is hard to detect and quantify 🤔



We don't know if the model was trained on the test set



Model weights are often unavailable



Hard to define a metric to detect leakage

We designed an experimental setup to detect memorization across models and bug benchmarks

LEAKAGE DETECTION

MEMBERSHIP

~ is my data part of The Stack ?

NEG. LOG LIKELIHOOD

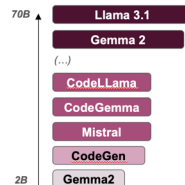
~ how surprised is the model by my input ?

5-GRAM ACCURACY

~ how close is the model output to the ground truth ?

MODEL SELECTION

9 open-source models



BENCHMARKS

5 Bug Benchmarks



New, likely unseen data

GitHub repos in Java and Python



Check out the paper! 😊

Our findings show that...

🤔 **Data leakage is an especially significant issue for Defects4J (V1.5)**

Smaller models, trained on far less data, seem more prone to memorization

Bigger and more recent models seem to exhibit lower memorization of benchmark solutions

If we are not careful with benchmark selection, we risk reporting inflated model performance due to data leakage

So, we suggest...



Evaluate on benchmarks with new data



Monitor leakage risk over time by computing membership, NLL and 5-gram matches



Carefully interpret benchmarks numbers

