Analyzing the Impact of Self-Admitted Technical Debt on the Code Completion Performance of Large Language Models

Lucas Witte[1]
Supervisor(s): Prof. Dr. Arie van Deursen[1],
Assistant Prof. Dr. Maliheh Izadi[1],
Ir. Jonathan Katzy,[1], **and Ir. Razvan Mihai Popescu**[1]

[1]EEMCS, Delft University of Technology, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 22, 2025

Name of the student: Lucas Witte
Final project course: CSE3000 Research Project
Thesis committee: Prof. Dr. Arie van Deursen, Assistant Prof. Dr. Maliheh Izadi, Ir. Jonathan Katzy, and Ir. Razvan Mihai Popescu, Associate Prof. Dr Avishek Anand

An electronic version of this thesis is available at http://repository.tudelft.nl/.

Analyzing the Impact of Self-Admitted Technical Debt on the Code Completion Performance of Large Language Models

Lucas Witte

Delft University of Technology
Delft, The Netherlands

Abstract

Large Language Models (LLMs) are increasingly integrated into development workflows for tasks such as code completion, bug fixing, and refactoring. While prior work has shown that removing low-quality data—including data smells like Self-Admitted Technical Debt (SATD)—can reduce model performance, the isolated effect of SATD at inference time remains unclear.

This study investigates the impact of SATD on LLM performance during code completion. Using The Heap dataset, we annotate over 5 million Java files with SATD bitmasks and construct a set of input–target pairs based on varying SATD contexts and masking strategies. Three code generation models, SmolLM2, StarCoder2, and Mellum, are evaluated on both comment and method generation tasks using standard text-based metrics and manual semantic classification.

Our results show that the presence of SATD in input has a negligible effect on generation quality. Instead, performance is primarily driven by target method length, structural complexity, and context size. We also find that metrics may misrepresent semantic correctness in the presence of non-functional elements such as comments. These findings suggest that careful control of input complexity is more critical than the presence of SATD alone when evaluating LLM performance on code.

Keywords

data smells, code generation, large language models, SATD

1 Introduction

Large Language Models (LLMs) have become widely used across various applications. As LLMs become more integrated into development tools and workflows, accurately evaluating their performance is crucial. Practitioners rely on these models for tasks like code completion, bug fixing, and refactoring. Understanding how input to a model can affect its behavior is essential to ensure its effectiveness and reliability in practice.

A key factor influencing model performance is the quality of training data. To ensure both efficiency and quality during training, datasets are typically preprocessed to address common data quality issues, such as duplicates, missing values, or outliers. In addition to these more apparent problems, recent work has identified a class of less obvious issues known as data smells [1][2]. In the context of code-related tasks, these smells are analogous to code smells in software engineering. One specific smell is self-admitted technical debt (SATD). This refers to comments in the source code where developers explicitly acknowledge technical debt (TD). TD is suboptimal code that is often introduced in rushed development, as quick and temporary fixes. The idea is that in the short term, these design choices are valuable, but it also creates debt since it needs to be paid off in the future. Common examples of SATD include comments marked with `TODO` or `FIXME`.

Recent work has shown that removing data smells, including SATD, from training data can reduce model performance [3]. However, this raises important questions about the nature and impact of such data. Specifically, little is known about the isolated effect of SATD on LLM performance, especially when SATD is provided at inference time as part of the input.

Moreover, understanding the impact of SATD is critical for the use and interpretation of benchmark datasets. If datasets used for evaluation contain data smells like SATD, they may artificially inflate or deflate model performance metrics depending on whether and how models leverage these artifacts. This can lead to misleading conclusions about model capabilities and hinder fair comparison. Thus, analyzing SATD's effect informs better dataset curation and evaluation strategies to improve the reliability and validity of benchmarking LLMs.

To bridge this gap, this study investigates the impact of SATD on code completion performance using LLMs. We base our analysis on The Heap [4], a dataset that consists of source code which has been decontaminated with respect to public training datasets, enabling a fair evaluation. Specifically, we will be answering the following research questions:

- **RQ1** *What is the presence of SATD in The Heap?*
- **RQ2** *What is the impact of SATD on the performance of LLMs for code completion tasks?*
- **RQ3** *Do models generate correct code, that doesn't match (broken) ground truth?*

To answer these questions, we perform an experimental study. First, we use a SATD detection tool to annotate The Heap with bitmasks indicating the presence of SATD. Based on these annotations, we construct input–target pairs for different scenarios and masking strategies. These inputs are fed to three code-generating LLMs: SmolLM2 [5], StarCoder2 [6], and Mellum [7], and its outputs are evaluated quantitatively and qualitatively.

Our results show that while SATD is present in a relatively small portion of The Heap dataset, it does not significantly impact the code completion performance of the evaluated LLMs. Instead, we find that other factors—such as target method length, structural complexity, and available context—play a much more substantial role in influencing generation quality. Furthermore, our qualitative analysis reveals that models do not recover from flawed or broken ground truth, and that automatic metrics may misrepresent semantic correctness when non-functional elements, such as comments, are present.

```
// TODO: implement proper login for admin
    users

// FIXME: this method is too long and needs to
    be refactored

// This is a workaround to ensure that the
    directory stream is always closed
```

Figure 1: Examples of SATD comments

These findings contribute to a deeper understanding of how contextual elements like SATD influence LLM behavior during inference. They also underscore the importance of careful experimental design when evaluating model performance. By isolating the effect of SATD, we provide practical insights for researchers and practitioners seeking to understand and improve the robustness of LLM-based code generation systems.

2 Background Research

Technical debt has been a topic of active research, particularly its impact on software quality and project management. Detection of technical debt (TD) can be valuable because it allows developers to estimate the extent and severity of the debt. With this information, they can go and fix the TD to prevent it from further hindering development. A particularly accessible form of TD is SATD, where developers explicitly acknowledge suboptimal code or design in comments. These comments can take many forms. Some illustrative examples are shown in Figure 1.

SATD is appealing for detection because it can be found directly by analyzing source code comments. Numerous tools have been developed to detect SATD, initially using pattern-matching approaches. For instance, a foundational study manually analyzed thousands of comments and identified 62 comment patterns that indicate SATD [8]. With the rise of machine learning and natural language processing, more advanced techniques have emerged, including text mining, deep learning models, and neural networks [9–11]. These newer tools often integrate additional data sources, such as issue trackers, pull requests, and commit messages, to enhance detection accuracy and provide a more comprehensive project-level view of technical debt.

Despite these advancements, one persistent challenge is the limited accessibility and reproducibility of SATD detection tools. A recent systematic literature review found that out of 68 tools, only 8 were publicly available, and even fewer were functional or easy to integrate into further research workflows [12].

Detection of SATD is also used to clean large datasets used to train LLMs [2]. SATD is not desirable since it contains suboptimal code that might negatively impact the performance of trained models. It has been shown that removing 12 categories of data smells, including SATD, before training code summarization models leads to a significant improvement on the performance of code summarization [3]. However, there has been no research on the impact of SATD in isolation, neither for training purposes nor for prompting purposes.

3 Methodology

To answer the research questions, a multi-stage pipeline is constructed, illustrated in Figure 2. We first detect SATD in The Heap (3.1). Then we generate input-target pairs (3.2), which are used by models to generate code (3.3). Finally, the generated code is evaluated both quantitatively and qualitatively (3.4). To support reproducibility, all code and data used in this pipeline, including intermediate results, are publicly available[1].

3.1 SATD Detection

SATD detection is performed in The Heap. This is a dataset that consists of source code that is not part of the training data for the LLMs we will be evaluating. This ensures a fair evaluation of the model's performance on unseen code.

There are three steps for SATD detection. First, the comments in a file are extracted using `tree-sitter`. Next, the comments are preprocessed by removing: license comments, Javadoc comments, and commented out code that do not contain a task annotation (i.e. `TODO`, `FIXME`, or `XXX`). This follows the preprocessing steps done to arrive at the dataset [13] used to train the SATD Detector [9], ensuring consistency between our input data and the training conditions of the model. Finally, we use the SATD Detector tool to identify the comments likely to contain SATD. It uses a text-mining approach, this method has been shown to outperform earlier pattern-matching techniques in both accuracy and recall.

During detection, a bitmask is generated for each file, where '0' indicates code not associated with SATD, and '1' marks code identified as SATD. These masks are stored as additional annotations in the dataset.

To answer *RQ1*, we perform a presence analysis of SATD in the annotated dataset. We quantify its prevalence both at the dataset and file level, measuring overall SATD frequency, the proportion of affected files, and SATD density per file.

3.2 Input and Target Generation

To evaluate the impact of SATD on LLM performance (*RQ2*) and their ability to generate semantically correct but non-matching code (*RQ3*), we construct various input–target pairs based on the SATD annotations. Each case specifies a masked target region and an input context constructed according to two masking strategies: *causal* and *Fill-in-the-Middle (FiM)*.

In causal masking, the input consists solely of tokens preceding the target, and the model predicts the next tokens in left-to-right order. In FiM masking, the input includes both prefix and suffix surrounding the target. FiM is especially useful for tasks such as inserting code between blocks or generating comments above methods, where both sides of the context are informative. FiM and causal masking use the same context window size. When the input exceeds this limit, we prioritize the code closest to the target. For causal masking, we truncate the start of the prefix. For FiM, we symmetrically truncate both prefix and suffix to preserve balance around the target.

We define 10 evaluation cases to study the effects of SATD on model behavior. These are grouped into three categories as summarized in Table 1. Each case is evaluated using both causal and

[1] https://github.com/lcwitteTUD/ML4SE-toolkit-SATD

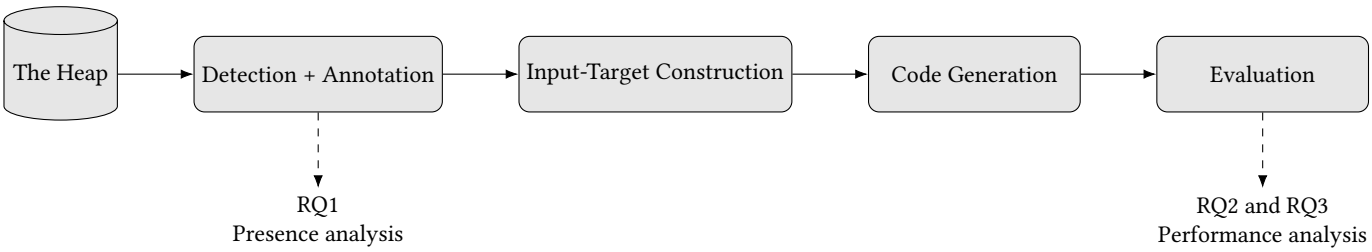

Figure 2: Overview of the methodology and its relation to the research questions

Table 1: Input-target cases grouped by research focus

Group	Case	Description
1	`in-smell`	Target is a SATD comment
	`no-smell-comment`	Target is a random comment from a file without SATD
2	`out-smell-distance{0,1,-1}`	Target is respectively the first, second or last method after the SATD
	`no-smell-method`	Target is a random method from a file without SATD
3	`multi(-1)-out-smell-{0,-1}`	Method after the last SATD in files with multiple SATD comments
	`prep-multi(-1)-out-smell-{0,-1}`	Same method, but all SATD removed from the file

FiM masking strategies, for a total of 20 configurations per model. The first group explores whether models reproduce SATD-like comments. The second group allows us to analyze whether proximity to SATD impacts model performance on method generation. The third group investigates whether removing SATD, by preprocessing the input, improves model performance, particularly in more realistic multi-SATD scenarios.

To ensure broad coverage while keeping generation costs feasible, we sample up to 4000 files for Group 1 (comment generation) and 1000 files for Groups 2 and 3 (method generation). However, the final number of generated examples per case may be lower, as some sampled files did not contain a valid method or comment matching the case criteria.

3.3 Code Generation

For evaluation, we selected three small language models: SmolLM2-135M, StarCoder2-3B, and MellumBase-4B. SmolLM2-135M uses only the causal masking strategy, whereas StarCoder2-3B and MellumBase-4B support both causal and Fill-in-the-Middle (FiM) masking. These models generate code given an input, and the generated outputs are compared against the target to evaluate performance.

To save computation time, generation is stopped early if repeated token patterns are detected, preventing infinite loops of repetitive output. We also limit the maximum number of tokens generated. This limit is set to the average length plus two times the standard deviation of target lengths, calculated separately for comments and methods using all examples in the dataset.

All code generations were performed on the DelftBlue Supercomputer [14], using NVIDIA V100 GPUs with 32GB memory. Each generation task used a single GPU. On average, comment generation cases (Group 1) required significantly less time than method generation cases (Groups 2 and 3). This is primarily due to the lower maximum token limit used when generating comments, leading to shorter and faster outputs. As a result, Group 1 could be evaluated over a larger set of files within the same compute budget. Overall, the full generation process across all models and cases took approximately 106 hours.

3.4 Evaluation

Quantitative evaluation. The generated code is quantitatively evaluated using standard text-based metrics widely adopted in code generation research (*RQ2*): Exact Match, Edit Distance, BLEU [15], METEOR [16], and ROUGE-L [17]. Exact Match calculates the percentage of generated outputs that exactly match the target code, providing a strict correctness measure. Edit Distance (also known as Levenshtein distance) quantifies the minimum number of single-character edits (insertions, deletions, or substitutions) needed to transform the generated code into the target, capturing how close the outputs are textually. BLEU (Bilingual Evaluation Understudy) measures the similarity between generated code and targets by calculating the n-gram precision. METEOR (Metric for Evaluation of Translation with Explicit ORdering) improves upon BLEU by incorporating stemming, synonymy, and considering word order. ROUGE-L measures the longest common subsequence between the generated text and the target. This offers another perspective on generation quality by focusing on sequence similarity rather than just n-gram overlap.

Qualitative evaluation. The generated code is qualitatively evaluated by examining 20 files for the method generation cases (*RQ3*, i.e., Groups 2 and 3). These files are selected to ensure that each includes generations for all relevant cases within the groupings (*out-smell*

Table 2: SATD presence statistics in the Java subset of The Heap

Measure	Value
Total source files	5,168,193
Total SATD comments	914,347
Files with ≥1 SATD comment	431,145 (8.34%)
Average SATD comments per file	0.18
Median SATD comments per file	0
Max SATD comments in a single file	2,230
SATD comments per KLOC	1.26

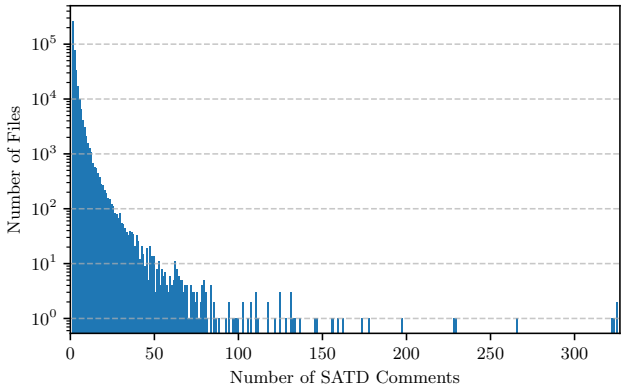

Figure 3: Distribution of SATD comments per file in the The Heap Java dataset

and *preprocessed-multi-out-smell*). Each generation is manually categorized into one of three classes based on its semantic alignment with the target:

(1) **Correct:** The generated method is functionally equivalent to the target.
(2) **Partial:** The method shares some semantics with the target (e.g., similar signature or partial body logic) but differs in completeness or intent.
(3) **Incorrect:** The method is functionally unrelated to the target (e.g., generates a different method, hallucinates code, or repeats context).

This classification enables us to better understand the types of generation failures and to assess whether low metric scores may still correspond to semantically valid outputs.

4 Results

In this section, we list the acquired results for every research question in order. That is, first the presence of SATD in The Heap (4.1), then the impact of SATD on the performance (4.2), and finally an analysis of the generated code (4.3).

4.1 SATD Presence in The Heap

Statistics for the SATD presence in The Heap are summarized in Table 2. Out of over 5 million source files, we detected a total of 914,347 SATD comments. Despite the large volume, SATD is sparsely distributed: only 8.34% of files contain at least one SATD comment, with an average of 0.18 SATD comments per file and a median of 0. Normalized by code size, we observe 1.26 SATD comments per thousand lines of code (KLOC). The maximum number of SATD comments observed in a single file was 2,230. This extreme outlier was manually verified. The results are based on a filtered set of SATD annotations in which common false positives, mostly from auto-generated files, were removed to improve accuracy. See subsection 5.1 for details.

Figure 3 shows the distribution of SATD comments per file in the Java subset of The Heap. For readability, this version omits files with zero SATD comments (91.66% of the dataset) and two extreme outliers containing 2,230 and 1,118 comments, respectively. After removing these, the highest remaining SATD count is 325, and the distribution still exhibits a long tail. Most SATD-containing files have fewer than 50 comments.

4.2 SATD Impact on the Performance of Code Completion

We present performance results across all models and evaluation cases in Figures 4 to 6, each corresponding to one of the three case groups defined in Table 1. Each figure includes two subplots showing BLEU and Edit Distance scores. In each subplot, boxplots for every case in the group are shown for each model setup, with mean values indicated by x-marks. We focus on these two metrics for brevity, as all five metrics (BLEU, Edit Distance, Exact Match, METEOR, ROUGE-L) exhibited consistent trends.

Due to a bug in suffix generation, FiM masking results were found to be invalid and are therefore excluded from this analysis. However, we include a detailed discussion of these invalid results, along with the full metrics, in Appendix A.1 and Appendix A.2, respectively.

Group 1: Reproduction of SATD-like Comments.
In the first group, which targets SATD and non-SATD comments, we observe that the *no-smell-comment* case consistently outperforms the *in-smell* case across all model setups.

Group 2: SATD Proximity in Method Generation.
In the second group, which evaluates method generation at varying distances from SATD, we find that performance improves as the distance from the SATD increases. Specifically, methods located further from the SATD (e.g., *distance = -1*) yield higher scores than those closer (e.g., *distance = 0*). Additionally, the *no-smell-method* case, which uses a random method from a SATD-free file, outperforms all out-smell cases for most models.

Group 3: Multi-SATD and Preprocessing Effects.
In the third group, which focuses on multi-SATD scenarios and preprocessing, two consistent trends emerge across all models and metrics. First, models perform better when generating the method furthest from the last SATD compared to the method immediately after it. Second, retaining SATD comments slightly improves performance over their removal, although the difference is modest.

Model Setup Performance.
When comparing model setups, *SmolLM2* exhibits the weakest performance across all groups and metrics. Both *StarCoder2* and

Table 3: Average target length, BLEU, and Edit Distance by classification

Classification	Target Length	BLEU	Edit Distance
Correct	126.17 ± 112.43	0.728 ± 0.301	0.883 ± 0.175
Partial	657.78 ± 703.15	0.288 ± 0.250	0.524 ± 0.261
Incorrect	581.16 ± 1288.66	0.027 ± 0.053	0.218 ± 0.159

MellumBase with causal masking perform substantially better, with similar results between them.

4.3 Manual Analysis of Generated Code

Figure 7 shows how the generated methods were classified for each case: as *Correct*, *Partial*, or *Incorrect*, based on semantic alignment with the target method. These results largely follow the same trends as the text-based metrics: cases with a higher proportion of semantically correct generations also tend to achieve better scores on BLEU and Edit Distance.

That said, there are clear examples where semantically correct generations still receive low metric scores. One such case is shown in Figure 8, where the generation is functionally equivalent to the target but differs due to missing non-functional comments. This leads to a low score despite the semantic match (it would be an exact match if comments were ignored).

We further observe that short targets tend to receive higher metric scores. This is often due to the frequent presence of common boilerplate such as `public void` in Java, which can lead to high n-gram overlap even in partially correct or incorrect generations. We verified this by calculating the average target character lengths for the three categories. As shown in Table 3, *Correct* generations had much shorter targets on average (126.17 characters) compared to *Partial* (657.78) and *Incorrect* (581.16). This difference is reflected in the text-based metrics.

In several cases, notably `no-smell` and `multi-out(-1)`, methods classified as *Correct* were also typically shorter and simpler than those in `out-smell(0)`. To better understand the relationship between method complexity and generation quality, we also analyzed target lengths and line counts per evaluation case. Boxplots showing the distribution of target lengths in characters and bar plots representing the proportion of methods with three lines or fewer are displayed in Figure 9. The mean values in the boxplots are marked with x-marks. For all groups except Group 1, we observe a clear correlation between metric scores and method complexity: simpler methods, characterized by shorter target lengths and a higher proportion of short methods, tend to receive higher scores.

For the preprocessing cases (`prep-*`), no major differences were found between generations with and without SATD removal. In many instances, the model produced near-identical outputs regardless of whether the SATD was present.

Finally, of the 40 analyzed files containing SATD comments (`out-smell` and Group 3 cases), only 6 had the SATD located directly above the method of interest. Of those, only two comments described TD that could be addressed in the corresponding method. In both cases, the model failed to resolve the issue and instead generated a incorrect method. In the remaining files, the SATD comments were located earlier in the file, typically in the body of the previous method, limiting their potential influence on the generation.

5 Discussion

This section reflects on our findings regarding the presence of SATD in The Heap dataset and its impact on code completion using small language models. First, we analyze the output of the SATD detector at scale and highlight the importance of validating automated annotations to ensure meaningful results (5.1). Then, we examine how SATD influences code completion performance and correctness, based on both quantitative metrics and manual evaluation (5.2). Together, these analyses offer insight into the limitations of current models and the importance of contextual factors beyond SATD alone.

5.1 SATD Presence in The Heap

The observation that most files contain no SATD comments (median = 0), and that the number of files decreases as the number of SATD comments increases, aligns with the general expectation that developers aim to minimize or avoid technical debt in their codebases.

However, a notable number of files exhibited extreme SATD counts: 11 files contained more than 10,000 SATD comments. To better understand these cases, we manually inspected the 100 files with the highest number of SATD annotations.

This investigation revealed that three specific comment patterns were responsible for disproportionately high SATD counts. The most significant involved the comment *"regression assertion (captures the current behavior of the code)"*, which appeared over 210,000 times across 17 files. These files were generated by the Randoop testing framework, which inserts this comment above each assertion. As these comments are automatically generated and descriptive in nature rather than indicators of technical debt, the corresponding annotations were excluded.

Two additional false positive patterns were identified: one involving repeated scientific documentation-style comments across multiple simulation files, and another involving blocks of commented-out string declarations containing the prefix `XXX`. These also did not represent self-admitted technical debt.

In total, annotations from 52 files were filtered, including 39 of the top 100 SATD-heavy files. These removals accounted for 215,879 SATD comments, approximately one-sixth of the original total. The results reported in Section 4.1 are based on this cleaned annotation set.

This case underscores the importance of validating automated SATD detection results, especially when analyzing large-scale datasets. Although the detector inevitably produces some false positives, we found that just three recurring patterns accounted for a substantial portion of the overall annotations.

5.2 Impact of SATD on Code Completion Performance and Correctness

Our results provide insight into the relationship between SATD and code completion performance, as well as the broader behavior of small language models in this task.

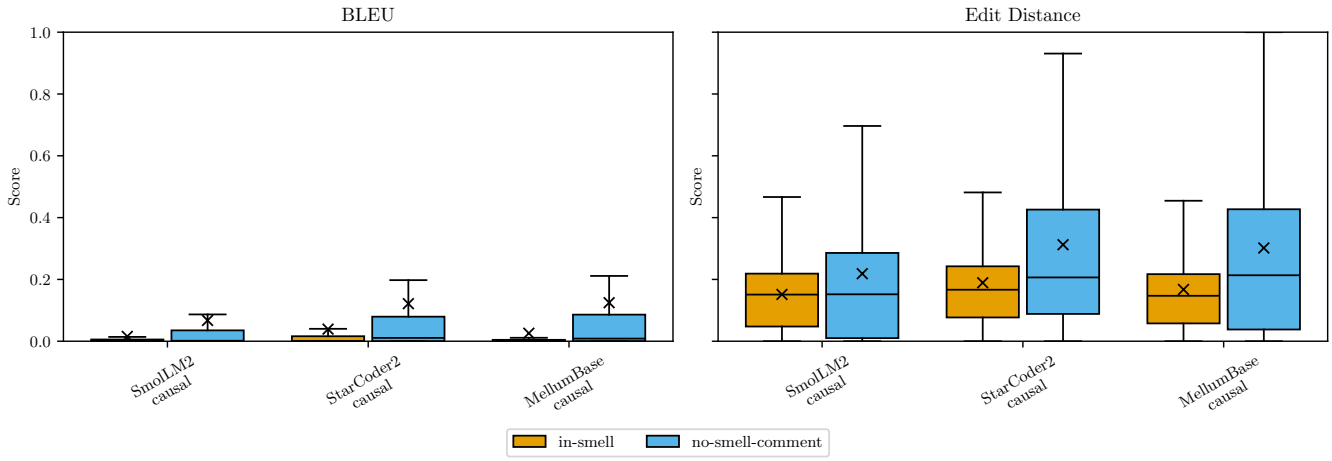

Figure 4: Metric scores for group 1

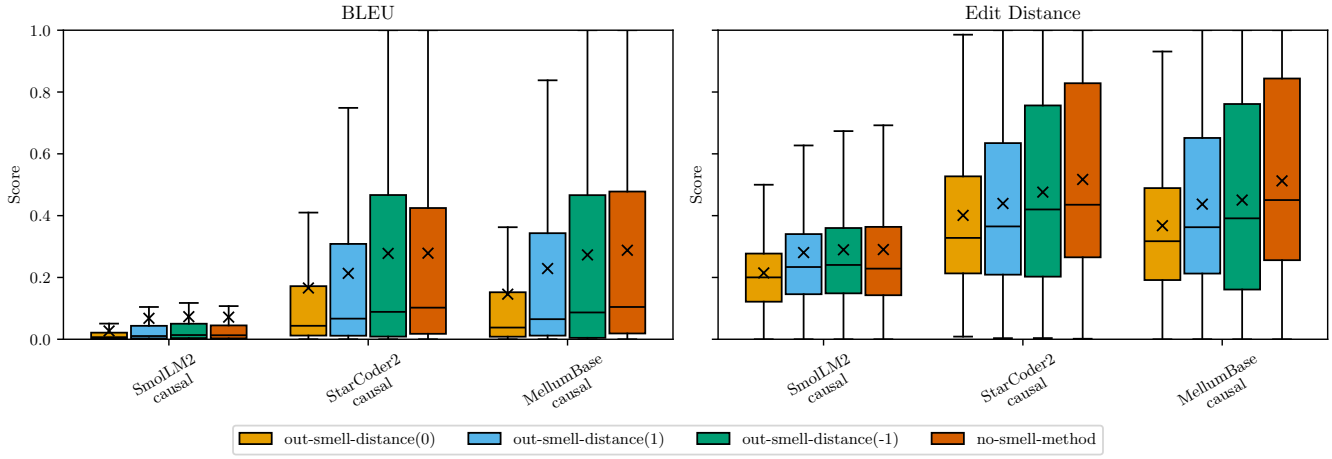

Figure 5: Metric scores for group 2

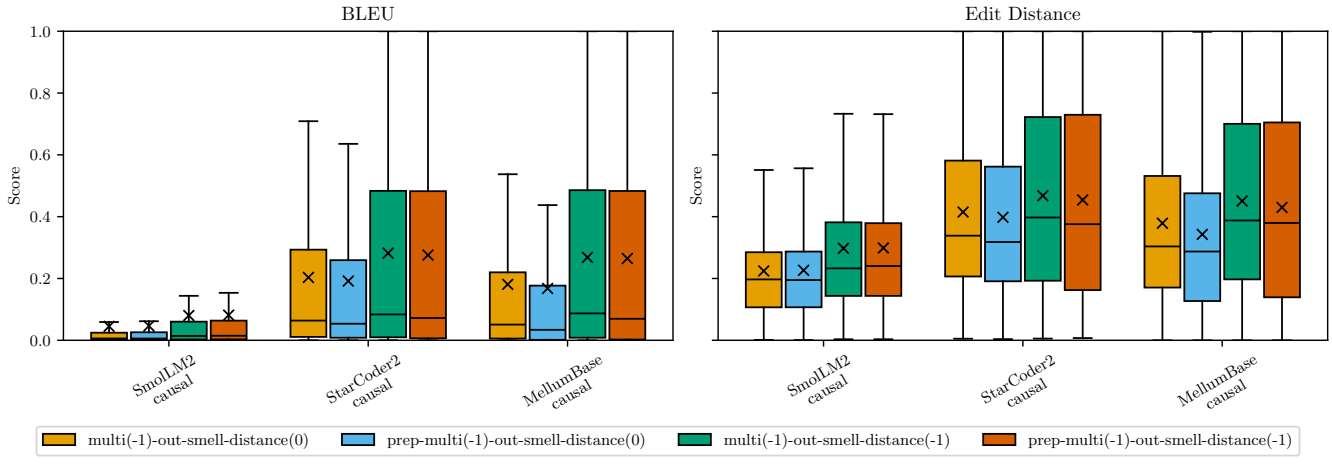

Figure 6: Metric scores for group 3

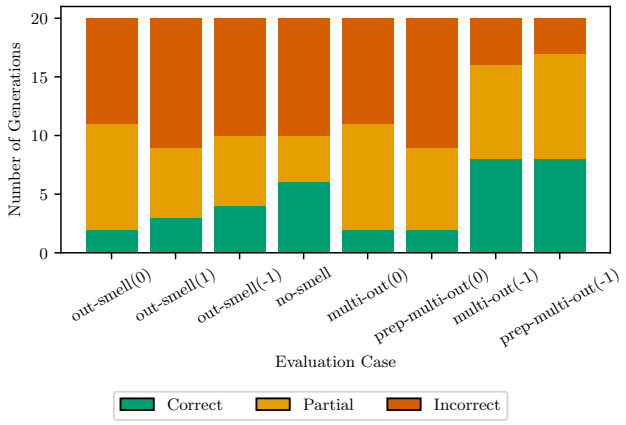

Figure 7: Qualitative evaluation of semantic generation correctness (causal only)

Case: `no-smell-method`
Model: `StarCoder2_3B (causal)`
Metrics: Edit Distance = 0.3416 BLEU = 0.0388

Target:

```
public HelpCtx getHelp() {
    // Show no Help button for this panel:
    return HelpCtx.DEFAULT_HELP;
    // If you have context help:
    // return new HelpCtx(
        SampleWizardPanel1.class);
}
```

Generation:

```
public HelpCtx getHelp() {
    return HelpCtx.DEFAULT_HELP;
}
```

Figure 8: Semantically correct generation example where metrics underestimate similarity

The clear trend in Group 2, where methods located further from SATD achieve higher scores, may be influenced by two distinct factors. First, we observed that methods further from SATD tend to be shorter, and shorter targets generally yield higher metric scores due to reduced complexity and greater n-gram overlap. Second, because the models use causal masking, later methods benefit from a larger context window that includes more preceding code. This increased context may aid generation quality independently of SATD proximity.

In contrast, the `no-smell` methods also achieve high scores, but without the influence of SATD or additional context. Here, the elevated scores are likely driven primarily by shorter target lengths and simpler method structures. This highlights the importance of controlling for target method complexity when evaluating model performance. When comparing different evaluation cases, the target methods should have similar average complexity to avoid introducing bias in the results.

In Group 3, we observed that removing SATD comments slightly reduced performance. However, since the targets remain identical between the preprocessed and unprocessed cases, this suggests that the SATD comments, though not directly part of the method, may still provide useful contextual signals for the model. Their removal thus leads to marginal performance degradation.

Taken together, these observations indicate that the presence of SATD itself does not significantly impact model performance. Instead, factors such as target length, method complexity, and available context dominate the observed trends. SATD appears to play a minimal role in determining output quality in this generation task.

As expected, the larger models, StarCoder2 and MellumBase, consistently outperform the smaller SmolLM2 across all groups and metrics. This aligns with general findings in LLM research, where increased parameter count correlates with stronger generation capabilities and better generalization.

Overall, metric scores correlate well with semantic classifications: higher BLEU and Edit Distance scores often correspond to semantically correct generations. However, this alignment is not perfect. Several examples show that functionally equivalent generations can receive low scores due to textual mismatches, particularly when the generation or target includes comments that the other does not. This exposes a key limitation of n-gram-based metrics when used to assess functional correctness. To address this, we recommend removing non-functional elements such as comments when the goal is to measure semantic similarity more accurately. Furthermore, this underscores the importance of complementing automatic metrics with qualitative analysis to better evaluate model performance and interpret the real-world implications of the generated outputs.

In our manual analysis, we encountered very few cases where the target method contained clearly broken or incomplete ground truth. In those few cases, the models did not manage to generate improved or correct replacements. This shows that when the target code is flawed, the models typically do not recover or correct the issues, but instead generate similarly flawed output.

6 Future Work

While this study provides initial insights into the influence of SATD on code completion performance, several promising directions remain for future research.

First, our evaluation focused on relatively small models due to resource constraints. A natural extension is to repeat the experiments with larger foundation models, such as CodeLlama [18] or GPT-based variants, to assess whether model scale amplifies or mitigates the observed trends.

Second, our analysis was restricted to Java. Expanding to other programming languages would provide a more comprehensive understanding of SATD's impact across all languages.

Third, our current design anchors generations at the method level. However, we observed that SATD comments often do not refer to the directly following method. Future studies could explore more localized or semantically linked contexts, such as generating

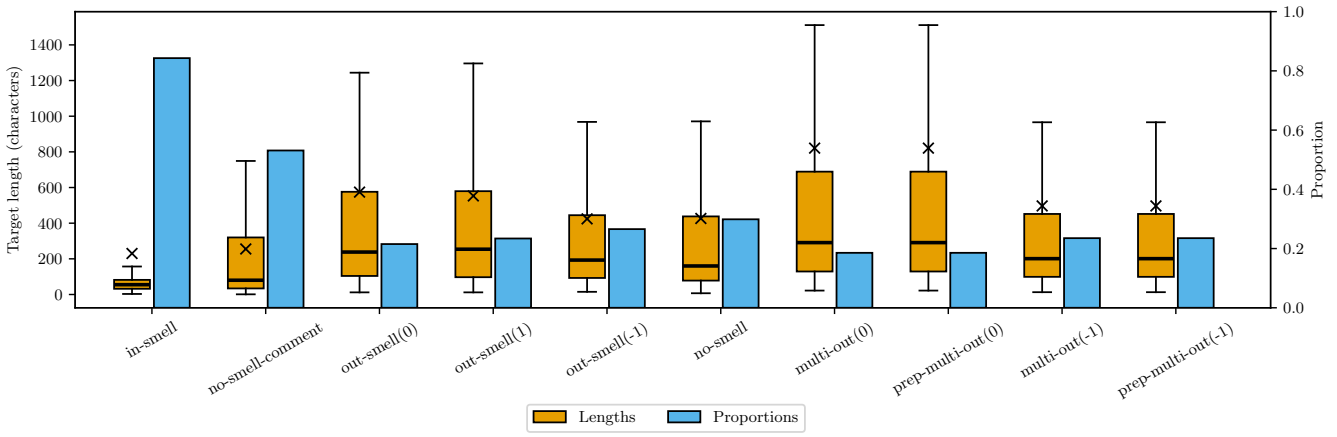

Figure 9: Lengths (in characters) and proportions of short methods (three lines or fewer) per case

code that directly relates to the comment itself (e.g., the block or line that immediately precedes or follows the SATD). This may allow a more targeted investigation into how models interpret and respond to SATD content during generation.

Fourth, our study focused exclusively on SATD as one instance of a broader class of data smells. Future work could extend this analysis to other unexplored data smells to understand how each uniquely affects LLM behavior. Systematically evaluating the impact of data smells on generation quality could guide data curation practices for training and evaluation datasets.

Finally, our FiM experiments were invalidated due to a bug where the suffix leaked into the target. While the results were excluded from the main analysis, re-running these experiments with corrected suffix boundaries would allow a proper comparison of causal and FiM masking strategies. This is especially important for evaluating the effect of out-smell scenarios, FiM would eliminate the factor of different context window sizes, allowing us to focus purely on the impact of the proximity of the SATD.

7 Conclusion

In this study, we investigated the impact of self-admitted technical debt (SATD) on the code completion performance of large language models (LLMs). To do so, we annotated the full Java subset of The Heap dataset, consisting of over 5 million files, with SATD bitmasks and constructed evaluation cases based on different SATD contexts and masking strategies.

Our analysis reveals that SATD is present in a small portion of the dataset. We found that the presence of SATD in the input has a negligible impact on model performance. Instead, generation quality is primarily influenced by factors such as target method length, structural complexity, and available context. Furthermore, Metric scores generally aligned with semantic correctness, but were sometimes misled by the presence of non-functional elements, such as comments. Finally, we found no evidence that models were able to generate correct completions in cases where the ground truth was broken.

Together, these findings suggest that while SATD may provide some contextual cues, it does not significantly affect code completion performance. Careful control over input context and target complexity remains more important than the presence of SATD alone.

8 Resposible Research

This study does not involve direct interaction with human subjects but relies on publicly available source code from The Heap dataset, which may include contributions from identifiable developers. The Heap includes files under weak and strong copyleft licenses and offers an opt-out mechanism via GitHub for developers who do not wish their code to be included. We only use this data for evaluation, not for training, to avoid legal and ethical concerns.

All code and data used to produce the results in this paper, including annotations, input–target pairs, generated completions, metric scores, and manually annotated generations, have been made publicly available through a GitHub repository. This aligns with FAIR data principles and makes every step reproducible. The models used (SmolLM2, StarCoder2, Mellum) are openly available at HuggingFace, though GPU access is required to replicate our experiments.

We took care to minimize bias in manual evaluation through clearly defined labeling criteria. Model selection may also influence results, but we mitigated this by evaluating multiple models of varying sizes. All results are reported transparently, and no data have been excluded or manipulated.

By highlighting the limited impact of SATD on LLM performance, this study contributes to a better understanding of model behavior and encourages more nuanced evaluation practices. We believe our work promotes the responsible development and deployment of code-generating models.

References

[1] Harald Foidl, Michael Felderer, and Rudolf Ramler. Data smells: Categories, causes and consequences, and detection of suspicious data in ai-based systems, 2022.

[2] Antonio Vitale, Rocco Oliveto, and Simone Scalabrino. A catalog of data smells for coding tasks. *ACM Trans. Softw. Eng. Methodol.*, 34(4), April 2025.

[3] Lin Shi, Fangwen Mu, Xiao Chen, Song Wang, Junjie Wang, Ye Yang, Ge Li, Xin Xia, and Qing Wang. Are we building on the rock? on the importance of data preprocessing for code summarization, 2022.
[4] Jonathan Katzy, Razvan Mihai Popescu, Arie van Deursen, and Maliheh Izadi. The heap: A contamination-free multilingual code dataset for evaluating large language models, 2025.
[5] Loubna Ben Allal, Anton Lozhkov, Elie Bakouch, Gabriel Martín Blázquez, Guilherme Penedo, Lewis Tunstall, Andrés Marafioti, Hynek Kydlíček, Agustín Piqueres Lajarín, Vaibhav Srivastav, Joshua Lochner, Caleb Fahlgren, Xuan-Son Nguyen, Clémentine Fourrier, Ben Burtenshaw, Hugo Larcher, Haojun Zhao, Cyril Zakka, Mathieu Morlon, Colin Raffel, Leandro von Werra, and Thomas Wolf. Smollm2: When smol goes big – data-centric training of a small language model, 2025.
[6] Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, Tianyang Liu, Max Tian, Denis Kocetkov, Arthur Zucker, Younes Belkada, Zijian Wang, Qian Liu, Dmitry Abulkhanov, Indraneil Paul, Zhuang Li, Wen-Ding Li, Megan Risdal, Jia Li, Jian Zhu, Terry Yue Zhuo, Evgenii Zheltonozhskii, Nii Osae Osae Dade, Wenhao Yu, Lucas Krauß, Naman Jain, Yixuan Su, Xuanli He, Manan Dey, Edoardo Abati, Yekun Chai, Niklas Muennighoff, Xiangru Tang, Muhtasham Oblokulov, Christopher Akiki, Marc Marone, Chenghao Mou, Mayank Mishra, Alex Gu, Binyuan Hui, Tri Dao, Armel Zebaze, Olivier Dehaene, Nicolas Patry, Canwen Xu, Julian McAuley, Han Hu, Torsten Scholak, Sebastien Paquet, Jennifer Robinson, Carolyn Jane Anderson, Nicolas Chapados, Mostofa Patwary, Nima Tajbakhsh, Yacine Jernite, Carlos Muñoz Ferrandis, Lingming Zhang, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. Starcoder 2 and the stack v2: The next generation, 2024.
[7] Nikita Pavlichenko, Iurii Nazarov, Ivan Dolgov, Ekaterina Garanina, Karol Lasocki, Julia Reshetnikova, Sergei Boitsov, Ivan Bondyrev, Dariia Karaeva, Maksim Sheptyakov, Dmitry Ustalov, Artem Mukhin, Semyon Proshev, Nikita Abramov, Olga Kolomyttseva, Kseniia Lysaniuk, Ilia Zavidnyi, Anton Semenkin, Vladislav Tankov, and Uladzislau Sazanovich. Mellum-4b-base, 2025.
[8] Aniket Potdar and Emad Shihab. An exploratory study on self-admitted technical debt. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 91–100, 2014.
[9] Zhongxin Liu, Qiao Huang, Xin Xia, Emad Shihab, David Lo, and Shanping Li. Satd detector: a text-mining-based self-admitted technical debt detection tool. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, ICSE '18, page 9–12, New York, NY, USA, 2018. Association for Computing Machinery.
[10] Irene Sala, Antonela Tommasel, and Francesca Arcelli Fontana. Debthunter: A machine learning-based approach for detecting self-admitted technical debt. In *Proceedings of the 25th International Conference on Evaluation and Assessment in Software Engineering*, EASE '21, page 278–283, New York, NY, USA, 2021. Association for Computing Machinery.
[11] Yikun Li, Mohamed Soliman, and Paris Avgeriou. Automatic identification of self-admitted technical debt from four different sources. *Empirical Software Engineering*, 28(3), April 2023.
[12] Edi Sutoyo and Andrea Capiluppi. Self-admitted technical debt detection approaches: A decade systematic review, 2024.
[13] Everton da S. Maldonado and Emad Shihab. Detecting and quantifying different types of self-admitted technical debt. In *2015 IEEE 7th International Workshop on Managing Technical Debt (MTD)*, pages 9–15, 2015.
[14] Delft High Performance Computing Centre (DHPC). DelftBlue Supercomputer (Phase 2). https://www.tudelft.nl/dhpc/ark:/44463/DelftBluePhase2, 2024.
[15] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, ACL '02, page 311–318, USA, 2002. Association for Computational Linguistics.
[16] Alon Lavie and Abhaya Agarwal. Meteor: an automatic metric for mt evaluation with high levels of correlation with human judgments. In *Proceedings of the Second Workshop on Statistical Machine Translation*, StatMT '07, page 228–231, USA, 2007. Association for Computational Linguistics.
[17] Chin-Yew Lin. ROUGE: A package for automatic evaluation of summaries. In *Text Summarization Branches Out*, pages 74–81, Barcelona, Spain, July 2004. Association for Computational Linguistics.
[18] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. Code llama: Open foundation models for code, 2024.

A Additional Results

This appendix presents extended results complementing the performance evaluation discussed in Section 4.2. We provide the remaining metric scores (A.1) and the invalidated FiM results (A.2).

A.1 Remaining Metric Scores

This appendix provides a complete overview of the remaining evaluation metrics not shown in the main paper. Specifically in Figures 10 to 12, we include plots for *Exact Match*, *ROUGE-L*, and *METEOR* across all case groups and model setups. These results complement the BLEU and Edit Distance trends discussed in subsection 4.2, and consistently reflect similar patterns in model performance.

A.2 FiM Evaluation Results (Invalidated)

This appendix contains performance results for all FiM-based input–target configurations. These results can be seen in Figures 13 to 15 are included for completeness, as they were part of the original experimental design.

Due to a bug in the suffix generation during FiM masking, the target was in many cases included in the input. This violates the non-leakage assumption and inflates performance. Therefore, the results presented here are invalid and should not be used for interpretation or comparison. Future work may revisit FiM-based evaluation with correctly constructed suffixes.

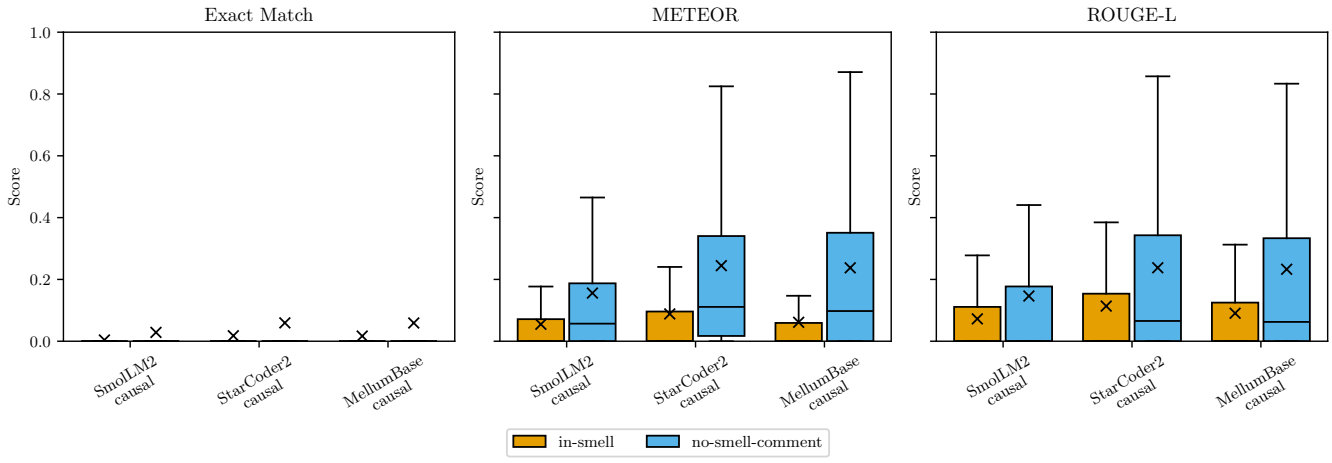

Figure 10: Metric scores for group 1

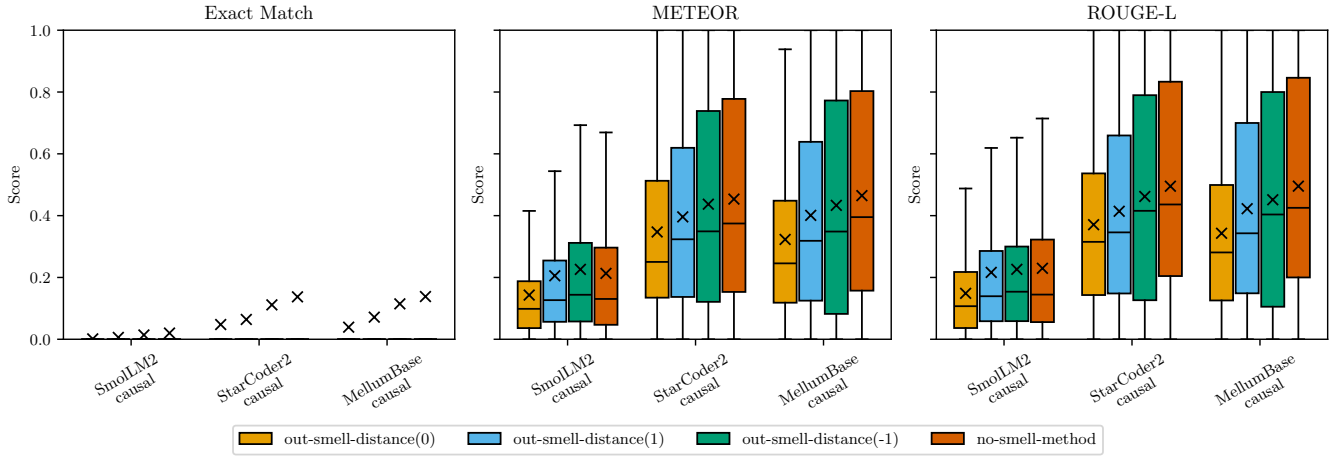

Figure 11: Metric scores for group 2

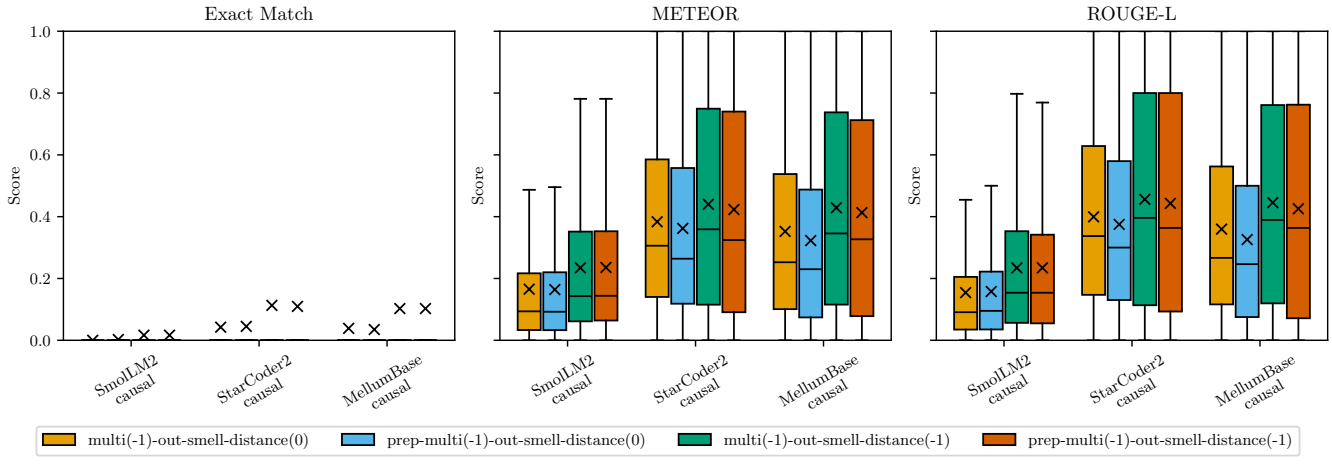

Figure 12: Metric scores for group 3

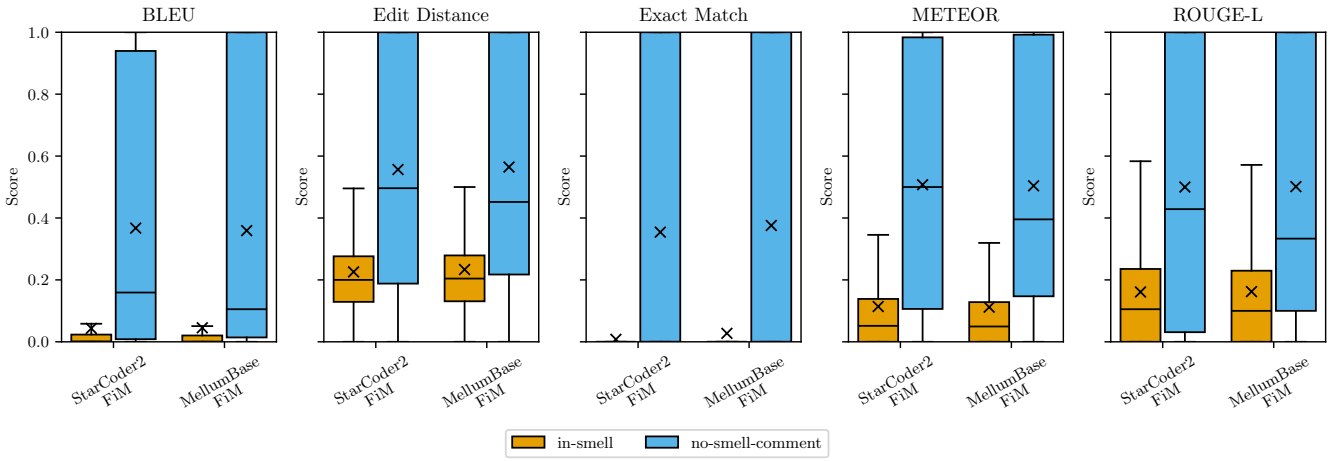

Figure 13: Metric scores for group 1

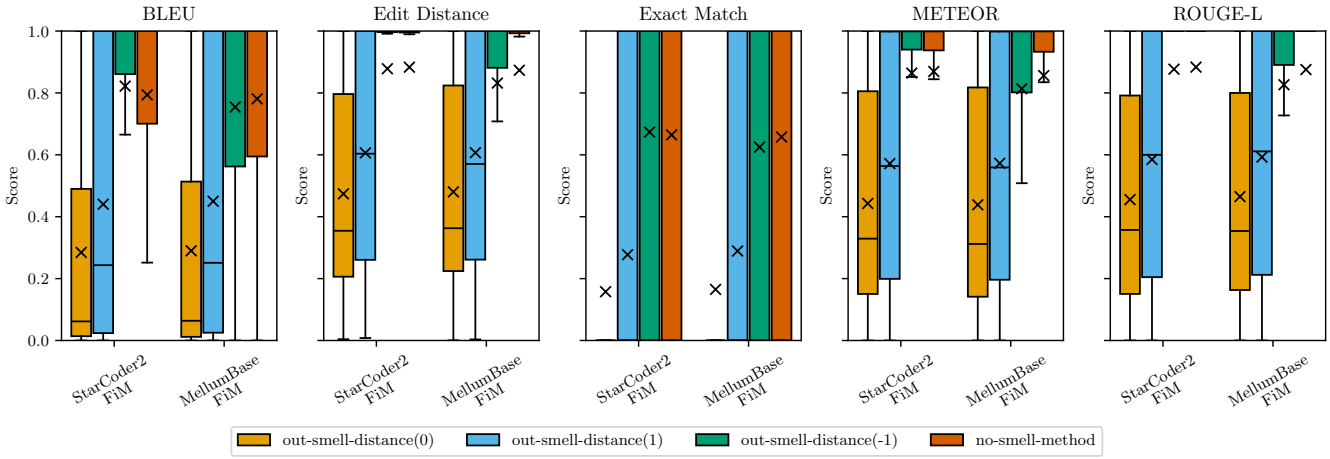

Figure 14: Metric scores for group 2

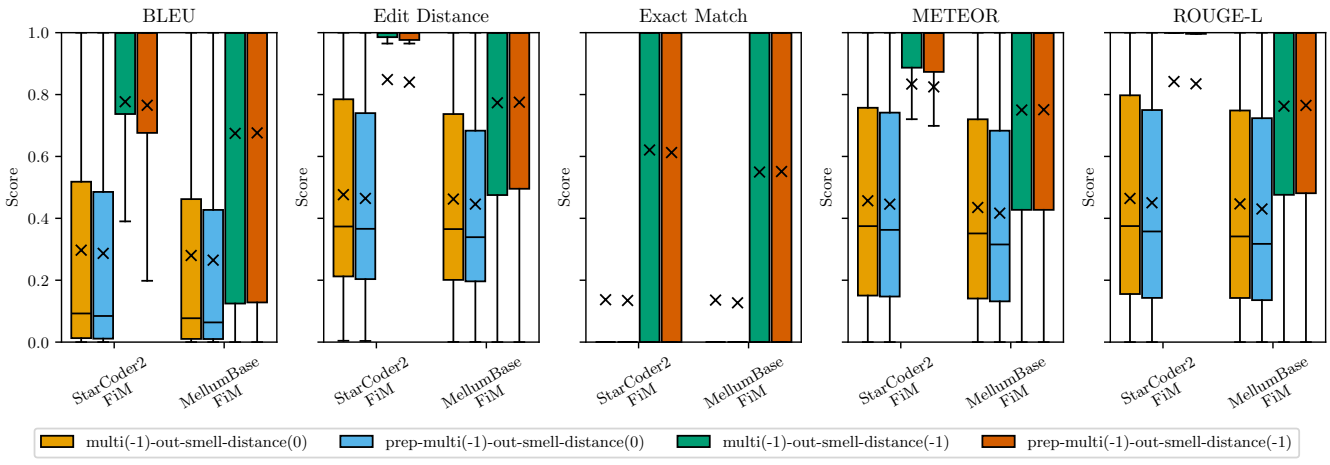

Figure 15: Metric scores for group 3