SACL: Understanding and Combating Textual Bias in Code Retrieval with Semantic-Augmented Reranking and Localization

Dhruv Gupta* Carnegie Mellon University dhruvgu2@andrew.cmu.edu

Gayathri Ganesh Lakshmy* Carnegie Mellon University gganeshl@andrew.cmu.edu

Yiqing Xie Carnegie Mellon University yiqingxi@andrew.cmu.edu

Abstract

Retrieval-Augmented Code Generation (RACG) is a critical technique for enhancing code generation by retrieving relevant information. In this work, we conduct an in-depth analysis of code retrieval by systematically masking specific features while preserving code functionality. Our discoveries include: (1) although trained on code, current retrievers heavily rely on surface-level textual features (e.g., docstrings, identifier names), and (2) they exhibit a strong bias towards well-documented code, even if the documentation is irrelevant. Based on our discoveries, we propose SACL, a framework that enriches textual information and reduces bias by augmenting code or structural knowledge with semantic information. Extensive experiments show that SACL substantially improves code retrieval (e.g., by 12.8% / 9.4% / 7.0% Recall@1 on HumanEval / MBPP / SWE-Bench-Lite), which also leads to better code generation performance (e.g., by 4.88% Pass@1 on HumanEval).

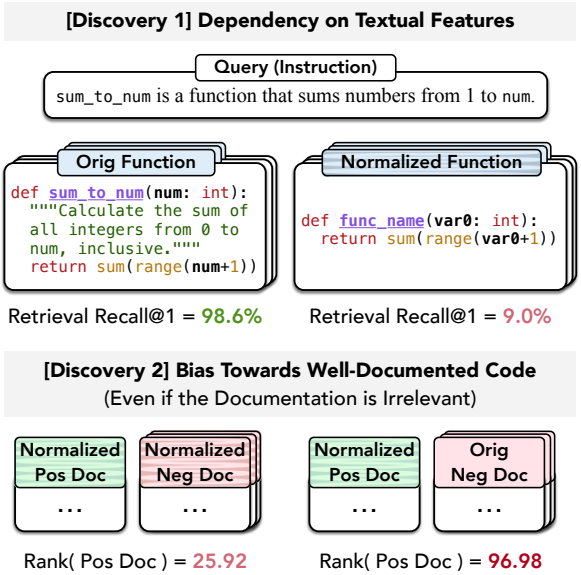


Figure 1: Summaries of our discoveries from the analyses. Our analyses reveal code retrievers' heavy dependence on textual features rather than functional semantics, leading to bias favoring well-documented code regardless of relevance.

1 Introduction

Retrieval-augmented code generation (RACG) is the technique of generating code based on relevant documents retrieved from a corpus (Koziolek et al., 2024; Lu et al., 2022). RACG is shown to be beneficial in script-level code generation, which provides background knowledge or functionally relevant snippets, and is particularly important for repository-level (repo-level) code generation, where models must be aware of other files within the repository (Wang et al., 2025). However, recent work has shown that retrieval quality remains a significant bottleneck for RACG performance (e.g., Agentless (Xia et al., 2024) only achieves 35.3% line localization accuracy on SWE-Bench).

While extensive analysis has been conducted on the capabilities and challenges of *text retrievers* (Dai et al., 2024; Karpukhin et al., 2020; Thakur et al., 2021), a systematic investigation of *code retrievers* remains relatively underexplored. The nature of code corpora differs fundamentally from text corpora due to their highly structured nature, with strict syntax rules and structures (Husain et al., 2019; Allamanis et al., 2018). Unlike text documents, the semantic meaning of a code snippet can be completely altered by small syntactic changes, making traditional retrieval analysis less effective for code search tasks. Such fundamental differences suggest that code retrievers may have significantly different behaviors from text retrievers, highlighting the need for more focused analysis.

In this work, to develop a deeper understanding of code retrieval, we conduct empirical analyses on both code retrievers and in-context rerankers to answer two critical research questions: **(RQ1) What features are code retrievers primarily based on?**

*Equal contribution.

Findings of the Association for Computational Linguistics: EMNLP 2025, pages 25052–25065
November 4-9, 2025

and **(RQ2) Do retrievers exhibit bias?** Specifically, we introduce a normalization-based analysis framework, where we systematically mask textual features such as docstrings, function names, and variable names or replace them with placeholders. Such transformations preserve the code's functionality but eliminate textual cues, allowing us to evaluate the dependence and bias of textual features.

We illustrate our discoveries in Figure 1. We observe that **[Discovery 1] although trained on code, current code retrievers exhibit strong dependency on textual features** (e.g., docstrings and function or variable names) and under-utilize the functionality of code. Specifically, when all textual features are normalized, we observe significant performance degradation on both embedding-based code retrieval and in-context code reranking. For instance, with normalization, the Recall@1 performance of GIST-large degrades from 98.6% to 9.0% on MBPP (Austin et al., 2021).

As shown in Figure 1, our analyses also reveal that **[Discovery 2] Retrievers consistently assign higher relevance scores to well-documented code, even when the documentation is functionally irrelevant**. Particularly, compared to Discovery 1's setting, where all the code documents are normalized, only normalizing the positive documents leads to even worse retrieval and reranking performances. The results indicate the bias towards well-documented code with meaningful identifier names, which may lead to preferring irrelevant but well-documented code over relevant but poorly documented code.

Based on these discoveries, we present SACL, which improves code retrieval with **S**emantic-**A**ugmented **C**ode Reranking and in-context **L**ocalization. Based on our discoveries that retrievers are more sensitive to textual features, in the reranking stage, we first generate textual descriptions for the retrieved code documents, and then aggregate the retrieval scores for the original code documents and textual descriptions for final re-ranking. Such design bridges the code and text modalities and mitigates the bias between well-documented and sparsely documented code. Based on the empirical discovery that in-context reranking also exhibits similar textual bias, for repo-level code generation, we further introduce semantic-augmented in-context localization, where we generate supplementary file descriptions for the repository structure to augment the context for file localization. Empirical results show that such methods are the most effective when the file names do not contain rich semantic information.

Our experimental results demonstrate significant improvements across three public benchmarks: HumanEval (Chen et al., 2021), MBPP (Austin et al., 2021), and SWE-Bench-Lite (Jimenez et al., 2023). For instance, SACL achieves 12.8%/9.4% code retrieval Recall@1 gain on HumanEval/MBPP under the full normalization setting and achieves 7.0% file localization Recall@1 on SWE-Bench-Lite with the Agentless pipeline (Xia et al., 2024). Our improvements on code retrieval and localization also leads to performance gain on code generation (e.g., 4.88% Pass@1 gain on HumanEval and 1.67% on SWE-Bench-Lite). These results highlight the effectiveness of our approaches in enhancing the semantic understanding capabilities of code retrievers and mitigating lexical bias.

2 Analysis: Textual Bias in Code Retrieval

This section answers our two research questions through two controlled experiments: **(RQ1)** What features are code retrievers primarily based on? and **(RQ2)** Do retrievers exhibit lexical-level bias? These experiments aim to identify whether code retrievers favor textual characteristics over functional semantics when matching queries to code.

2.1 RQ1: What Features are Code Retrievers Based on?

Setup. We quantify the importance of various features through a controlled study, where we progressively replace surface-level code features with dummy placeholders (e.g., "func_0", "var_0"). We call the process **"normalization"**. Specifically, we compare the Recall@1 performance of various code retrievers in five normalization settings:

① no normalization (i.e., the original code). ② removing docstrings and comments, ③ renaming function names, along with removing docstrings, ④ renaming variable names, along with removing docstrings, and ⑤ renaming both function names and variable names, along with removing docstrings (i.e., the combination of ②~④).

Note that this study preserves functional equivalence while enabling controlled ablation of specific features with rich textual information.

We study two categories of methods using four models: embedding-based retrievers (GIST-large, TE3-small), which rank the code documents based

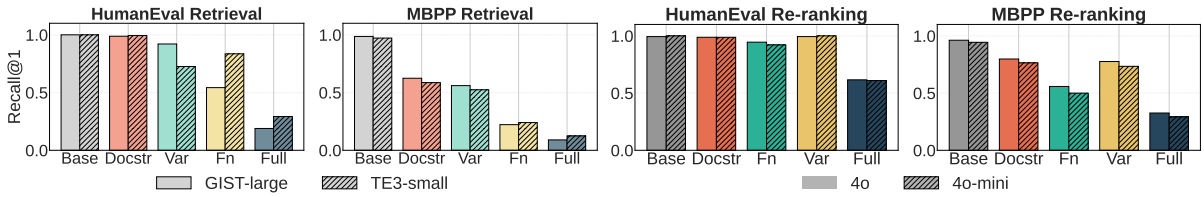


Figure 2: Impact of code normalization techniques on retrieval performance (Recall@1) across datasets with different embedding-based retrievers (*left*) and in-context rerankers (*right*). The results consistently show that all normalization techniques reduce retrieval effectiveness, with function name normalization and full normalization having the most significant negative impact.

on the cosine similarity with the query, and LLM-based rerankers (GPT-4o, 4o-mini), where we provide the ground truth (GT) document and the top-50 documents retrieved under the "no normalization" setting in the context, and then prompt the LLM to identify the most relevant document.

Results. As shown in Figure 2, results on HumanEval and MBPP reveal that both embedding-based retrievers and LLM-based rerankers have significant performance degradation under the normalization settings, especially the full normalization setting. This indicates that they heavily rely on textual features in retrieval or reranking.

We also observe that different models exhibit distinct sensitivities to different normalization methods. For instance, GIST-large shows more severe performance degradation with function name normalization (54.3% to 18.9%), while TE3-small suffers greater relative impact when variable names are normalized. Notably, docstring removal impacts MBPP significantly more than HumanEval (37% drop in MBPP versus minimal decrease in HumanEval). This difference stems from HumanEval's natural language descriptions containing function signatures that exactly match corpus signatures, providing strong retrieval signals even without docstrings, while MBPP's queries have fewer direct lexical matches.

The main discovery of this analysis is that **[Discovery 1] retrievers heavily rely on textual features, including docstrings and identifier names, rather than deeper semantic information such as the functionality of the code**. One possible explanation is that among the contrastive pairs used for retriever training (e.g., docstring-function pairs and StackOverflow QA pairs) (Husain et al., 2019), the textual queries have a high degree of lexical overlap with docstrings, function names, variable names, etc., and such correlation is captured by the retriever model.

Embedding Model: GIST-large				
Normalization Type			(**S1**) Norm GT &	(**S2**) Norm GT &
Docstring	Var	Func	Norm Others	Orig Others
✗	✗	✗	1.00	1.00
✓	✗	✗	1.01	1.07
✓	✓	✗	1.35	5.18
✓	✗	✓	8.05	20.29
✓	✓	✓	87.18	288.27
Embedding Model: OpenAI/text-embedding-3-small				
Normalization Type			(**S1**) Norm GT &	(**S2**) Norm GT &
Docstring	Var	Func	Norm Others	Orig Others
✗	✗	✗	1.00	1.00
✓	✗	✗	1.01	1.04
✓	✓	✗	2.23	22.48
✓	✗	✓	1.74	1.95
✓	✓	✓	25.92	96.98

Table 1: Average Rank of the GT document (↓) on the HumanEval dataset. The results demonstrate that when only the ground truth document is normalized (S2) while others remain in their original form, the GT document's rank deteriorates dramatically compared to when all documents are normalized (S1). Such results reveal a strong bias toward textual features over semantic relevance.

2.2 RQ2: Do retrievers exhibit bias?

Setup. Following §2.1, we further investigate whether code retrievers have a bias towards code containing more or fewer textual features (e.g., docstrings and function/variable names). Towards this goal, unlike the setting in §2.1 (**S1**), where all the documents are normalized, we introduce an asymmetric normalization setting (**S2**), where only the ground truth (GT) document for each query is normalized, while the remainder of the corpus is left in its original form. The comparison of the retrievers' performances under S1 and S2 allows us to assess whether models penalize stylistic deviations in semantically equivalent code.

Results. As shown in Table 1, the retrievers' performance further decreases when the irrelevant code documents are more well-documented than the ground truth one. For instance, in the most ex-

treme case where all the identifiers and docstrings are normalized, the rank of the GT document jumps from 87.18 to 288.27 for GIST-large retrieval (and from 25.92 to 96.98 for TE3-small).

This degradation reveals a clear inductive bias in current retrieval models: in many cases, the retriever assigns a higher rank to irrelevant but well-documented code over the semantically correct, normalized gold document. In other words, we observe that **[Discovery 2] retrievers tend to assign higher scores for well-documented code with meaningful identifier names, even if the documentation is irrelevant to the query**.

3 Methodology

Both **[Discovery 1]** and **[Discovery 2]** reveal that code retrievers heavily rely on textual information (e.g., documentation and identifier names) rather than understanding of code structure. To combat this issue, SACL introduces two techniques: semantic-augmented code reranking and semantic-augmented in-context localization. Both methods augment the retrieved code or structure with textual descriptions to improve the encapsulation of semantic information.

3.1 Semantic-Augmented Code Reranking

Traditional retrieval systems often struggle with the semantic gap between natural language queries and code documents.

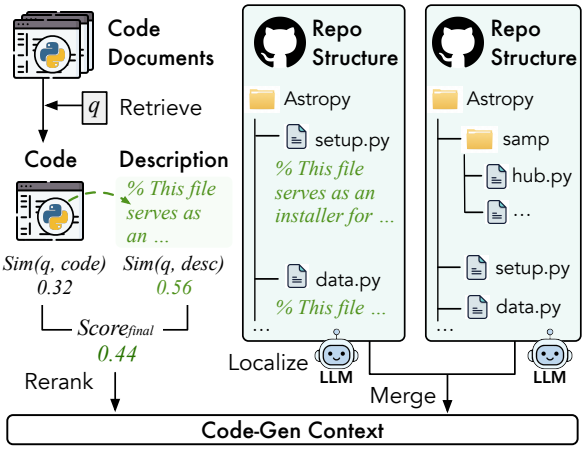
To bridge this gap, we enhance the re-ranking process with semantically rich descriptions. After retrieving the initial top-k code documents, we prompt an LLM to generate concise natural language descriptions of each code snippet's functionality and purpose. These descriptions provide an alternative representation of the code that emphasizes semantic content over syntactic structure.

Then we combine the relevance scores between (1) the original code documents and the queries, and (2) the textual descriptions and the queries:

$$Score_{final} = (1-\alpha) \cdot Score_{code} + \alpha \cdot Score_{desc} \quad (1)$$

where $\alpha \in [0, 1]$ is a tunable hyperparameter controlling the influence of each score component.

This approach transforms the cross-modal comparison problem (text-to-code) into a more tractable text-to-text comparison, enabling more semantically meaningful ranking of code documents based on natural language queries.



(left) Semantic-Augmented Code Reranking

(right) Semantic-Augmented in-context Localization

Figure 3: Flowchart illustrating our two main approaches. *(left)* Documents are retrieved based on query similarity, then augmented with generated descriptions. The final ranking score combines both code-query and description-query similarity scores to improve retrieval performance. *(right)* Repository structure is enhanced with descriptive summaries for each file (shown in green italics). This augmented structure significantly improves the LLM's ability to localize relevant files for code modification tasks.

3.2 Semantic-Augmented in-context Localization

Repository-level code generation required comprehensive understanding of code repository structure to effectively navigate and modify complex codebases (Xia et al., 2024). When previous works such as (Xia et al., 2024) uses repository structures to localize code that needs editing, they present the structure in a hierarchical format that represents the directory and file organization, as illustrated in Figure 3.

Building on our discoveries that retrievers tend to focus on textual information rather than functional semantics, we aim to augment structural representations with rich textual descriptions to improve localization performance. We enhance the standard repository structure by generating brief semantic descriptions for each file's contents in the repository. The descriptions are generated by prompting an LLM to analyze each file individually and summarize its contents (classes, functions, etc.), purpose and functionality in a couple sentences. The augmented repository structure serves as input to an LLM tasked with identifying potentially suspicious files related to a reported bug or issue. These semantic descriptions allow the

model to better understand file functionality and relationships, improving subsequent localization steps by reducing noise. To improve efficiency in inference, we generate descriptions using a small (Llama-3.1-8B-based) file summarization model, which already shows significant performance gain in file localization.

4 Experiments

With our experiments, we aim to answer the following research questions: **(RQ1)** What impact does SACL have on Code retrieval performance? **(RQ2)** Given this code retrieval performance, what is the downstream code generation performance improvement? **(RQ3)** Why does semantic augmentation benefit code retrieval? **(RQ4)** Which hyper-parameters are optimal?

4.1 Experiment Setup

Datasets. We evaluate our approach on three widely used benchmarks: HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021) are script-level algorithm problem datasets and SWE-Bench-Lite (Jimenez et al., 2023) is a repo-level issue-solving dataset. For HumanEval and MBPP, we evaluate under various "normalization" settings (as introduced in §2), which preserve the functionalities of the code documents but are more challenging to retriever models.

Evaluation Metrics. We follow existing work (Wang et al., 2025) and report Recall@k (k=1,5,10) for code retrieval and report Pass@1 for code generation. We additionally evaluate file and line localization accuracy (Xia et al., 2024) for SWE-Bench-Lite, which checks whether the corresponding generated patch edits a superset of all locations in the ground truth patch.

Implementation Details. For scalability concerns, we only generate a short description (under 100 words) for the code documents using a small model (Llama-3.1-8B-Instruct). For SWE-Bench Lite, we integrate our approach into the file localization step of the Agentless pipeline (Xia et al., 2024), which prompts an LLM to identify relevant files based on the repository structure in the format of a tree.

We provide more experimental details in A.1.

4.2 Code Retrieval Results

Script-level Code Generation Results. Table 2 presents code retrieval results under HumanEval and MBPP across different normalization settings, helping us answer **(RQ1)**. We observe that SACL demonstrates significant improvements over the baseline (e.g., improving Recall@1 for up to 15.2% on HumanEval and 14.8% on MBPP). Particularly, under the most challenging setting where all textual features are normalized, SACL still obtains substantial performance gain, which indicates that LLMs can still effectively summarize the functionalities even if all textual features are normalized. Our approach leverages LLM's strong code understanding property to compensate code retriever's bias toward textual features and hence effectively capture the semantic meaning of code.

In addition to the base model, we further compare SACL with GAR (Li et al., 2024; Mao et al., 2020), a query augmentation method that first uses the generated outputs to augment the query and then conducts retrieval and code generation. As shown in Table 3, all the methods are nearly perfect with docstring normalization, and SACL significantly outperforms GAR by 17.1% and 9.1% Recall@1 under function name and full normalization settings, demonstrating that document augmentation is more effective than query augmentation for normalized code retrieval.

Repo-level Issue-Solving Results. As shown in Table 4, SACL achieves significant performance gain on fault localization on SWE-Bench-Lite. For instance, we improve the file localization accuracy by 8.0%/7.0% for 4o-mini/GPT-4o. The consistent performance gain across different models highlights the effectiveness of augmenting repository structures with richer contextual information for fault localization. Specifically, as shown in later analysis (§4.4), the file descriptions may contain high-level descriptions of the file's purpose, its relationship to other files, or its utility to the whole repository, which are neglected in the repository tree structure. Such information reveals the high-level role and interconnections of files in the repository, which are relevant to the issues.

4.3 Code Generation Results

Script-level Code Generation Results. Results in Table 5 demonstrate that our performance gain in retrieval also translates to the improvement in code generation for various code generation models, highlighting the robustness of our method, tackling our **(RQ2)**. On MBPP, which presents a more challenging scenario due to lower lexical overlap between queries and relevant code (as shown in Figure 4), we observe improvements across all

Norm. Type	HumanEval Recall@1			HumanEval Recall@5			HumanEval Recall@10			MBPP Recall@1			MBPP Recall@5			MBPP Recall@10		
	Base	SACL	Δ	Base	SACL	Δ	Base	SACL	Δ	Base	SACL	Δ	Base	SACL	Δ	Base	SACL	Δ
Docstring	98.8	98.8	0.0	100.0	100.0	0.0	100.0	100.0	0.0	62.4	70.2	↑7.8	87.0	89.2	↑2.2	91.4	93.2	↑1.8
Func Name	54.3	69.5	↑15.2	77.4	86.0	↑8.6	83.5	89.6	↑6.1	22.0	36.8	↑14.8	39.8	51.0	↑11.2	49.0	55.0	↑6.0
All	18.9	31.7	↑12.8	34.1	43.3	↑9.2	42.7	46.3	↑3.6	9.0	18.4	↑9.4	18.6	29.0	↑10.4	23.6	31.4	↑7.8

Table 2: The Recall@k retrieval performance of the baseline and SACL under different normalization settings. We use GIST-large as the retriever and Llama-3.1-8B-Instruct for generating descriptions. We highlight results showing SACL > *Base* with green (darker green when having 5%+ increases or perfect results).

Norm. Type	HumanEval Recall@1			
	Base	GAR	SACL	Δ
Docstring	98.8	99.4	98.8	↓0.6
Func Name	54.3	52.4	69.5	↑17.1
All	18.9	22.6	31.7	↑9.1

Table 3: Comparison of SACL with GAR (Li et al., 2024; Mao et al., 2020) on HumanEval. We use GIST-base as the retriever and Llama-3.1-8B-Instruct for code generation (GAR) and description generation (SACL). The final column shows SACL's advantage over GAR.

Method	Localization Accuracy	
	Line	File (Δ)
SWE-Agent (GPT-4o)	29.3	58.3
AppMap Navie (GPT-4o)	29.7	59.7
AutoCodeRover (GPT-4)	29.0	62.3
Moatless (GPT-4o)	36.0	73.0
Agentless (GPT-4o-mini)	32.7	70.0
+ SACL	34.7 (↑2.0)	78.0 (↑8.0)
Agentless (GPT-4o)	40.0	79.0
+ SACL	**42.3** (↑2.3)	**86.0** (↑7.0)

Table 4: Fault Localization results on SWE-Bench-Lite, compared to other baselines. We compute the % of instances where the retrieved/LLM-localized files/lines contain the fault location. File-level localization is computed using the list of potential files identified by agentless at the end of the localization phase. Line-level localization is computed using final patches after testing and re-ranking.

normalization settings. Note that under the most challenging setting of full normalization, where even providing the normalized GT documents in the context only gives marginal code generation performance gain, SACL still delivers an improvement of 2.2 Pass@1. These results demonstrate that better retrieval directly translates to improved generation performance, with the benefits being most pronounced in scenarios where code lacks rich textual features.

Repo-level Issue-Solving Results. As shown in Table 6, our semantic-augmented approach also improves issue-solving rates on SWE-Bench-Lite (e.g., by 1.7% Pass@1 over Agentless (GPT-4o)).

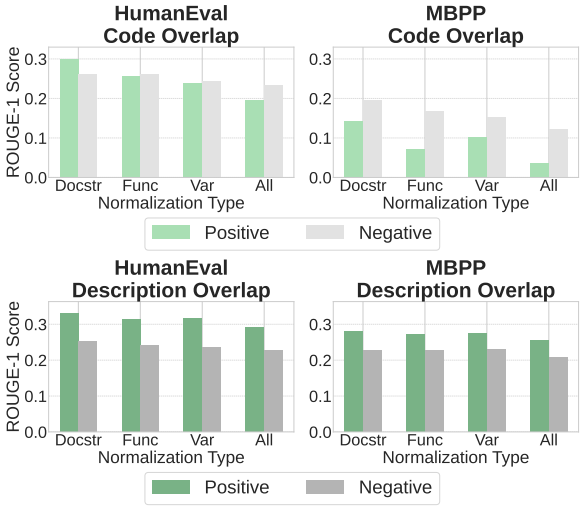


Figure 4: Lexical overlap (ROUGE-1 scores) between the query and the positive/negative code/descriptions in HumanEval and MBPP. We show the negative example with the highest overlap with the query. While under the normalization conditions, the negative code has similar or even higher lexical overlap with the query than the positive one, the positive descriptions always have higher ROUGE-1 scores.

This is consistent with previous work's observation that fault localization accuracy is a bottleneck for repo-level code generation (Xia et al., 2024). The improvement in resolved issues is particularly crucial as it further validates our approach's effectiveness in practical software engineering scenarios.

4.4 Performance Analysis

SACL Improves Query-Doc Lexical Overlap. To investigate **(RQ3)**, we hypothesize that semantic descriptions enhance code retrievers by improving lexical overlap between the query and documents, especially when the surface-level features in code are normalized. To test our hypothesis, we compare the ROUGE scores between the query and the positive and negative examples.

Results in Figure 4 reveal that the lexical overlap between positive and negative descriptions is larger than that of code documents. Particularly, on the

Normalization Type	HumanEval Pass@1						MBPP Pass@1					
	Qwen2.5-Coder-7B			Deepseek-coder-7b			Qwen2.5-Coder-7B			Deepseek-coder-7b		
	Base	SACL	Δ	Base	SACL	Δ	Base	SACL	Δ	Base	SACL	Δ
Docstring	99.39	99.39	0.00	99.39	99.39	0.00	57.40	61.60	↑4.20	57.40	61.60	↑4.20
Func Name	99.39	99.39	0.00	99.39	99.39	0.00	19.00	24.40	↑5.40	19.00	24.40	↑5.40
All	93.29	98.17	↑4.88	93.29	98.17	↑4.88	6.80	9.00	↑2.20	6.80	9.00	↑2.20

Table 5: The Pass@1 performance of the baseline and SACL on HumanEval and MBPP.

Method	% Resolved
SWE-Agent (GPT-4o)	18.3
AppMap Navie (GPT-4o)	21.7
AutoCodeRover (GPT-4)	19.0
Moatless (GPT-4o)	24.7
RepoUnderstander (GPT-4)	21.3
Agentless (GPT-4o-mini)	14.7
+ SACL	16.0 (↑1.3)
Agentless (GPT-4o)	24.3
+ SACL	**26.0** (↑1.7)

Table 6: Code generation results on SWE-Bench-Lite.

Category	Frequency (%)	Gain (%)
Functional Purpose	300 (100.00%)	↑24 (8.00%)
Core Components	242 (80.67%)	↑20 (8.26%)
File Relations	42 (14.00%)	↑4 (9.52%)

Table 7: Analysis of the file descriptions used by SACL. We manually design the categories based on the descriptions' content and use 4o-mini for categorization.

challenging MBPP dataset, the positive document's lexical overlap with the query is on average lower than that of the best negative document under the normalization settings, while descriptions maintain a clear separation between positive and negative examples even under full normalization.

SACL Enriches Context with File Semantics. To help understand why our in-context localization method is effective for repo-level code generation, we analyze the contents of generated descriptions for the GT files in SWE-Bench-Lite. Particularly, we define three categories for the descriptions' content: (1) Functional Purpose - the file's overall functionality; (2) Core Components - specific functions, classes, or data structures; and (3) File Relationships - connections to other repository files. Then we use GPT-4o-mini to categorize each description of the GT files.

Answering **(RQ3)**, our analysis shows that file descriptions serve multiple purposes simultaneously. As shown in Table 7 all descriptions (100%) cover Functional Purpose, 80.67% describe Core Components, and 14.00% mention File Relationships. This semantic enrichment contributes to the overall 8% gain in file-level localization, demonstrating how augmented context helps models better understand file purposes and interrelationships within repositories. Within the core components and file relations categories, we observe a respective 8.26% and 9.52% gain. Both are slightly higher than the overall gain.

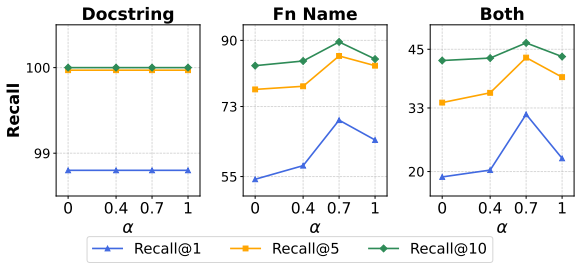


Figure 5: Impact of normalization parameter α on Recall performance. The three plots show performance across different normalization approaches: Docstring (left), Function Name (middle), and Both - Functions and Variables (right).

Hyper-Parameter Analysis. To answer **(RQ4)**, we illustrate the impact of the weighting parameter α across different normalization approaches. As shown in Figure 5, for all Recall@k values (k=1, 5, 10), the performance consistently peaks at α=0.7, except in the case of docstring normalization, where all methods perform nearly perfectly. In contrast, pure-description retrieval (α=1) and pure-code retrieval (the baseline, with α=0) both show lower performance. This result indicates that SACL achieves a balance between retrieval based on code snippets and descriptions, where code preserves full information and descriptions capture the semantics that may be challenging to encode from the code.

4.5 Case Studies

We present two representative case studies to illustrate how SACL effectively bridges the semantic gap between natural language queries and code, especially under normalization settings.

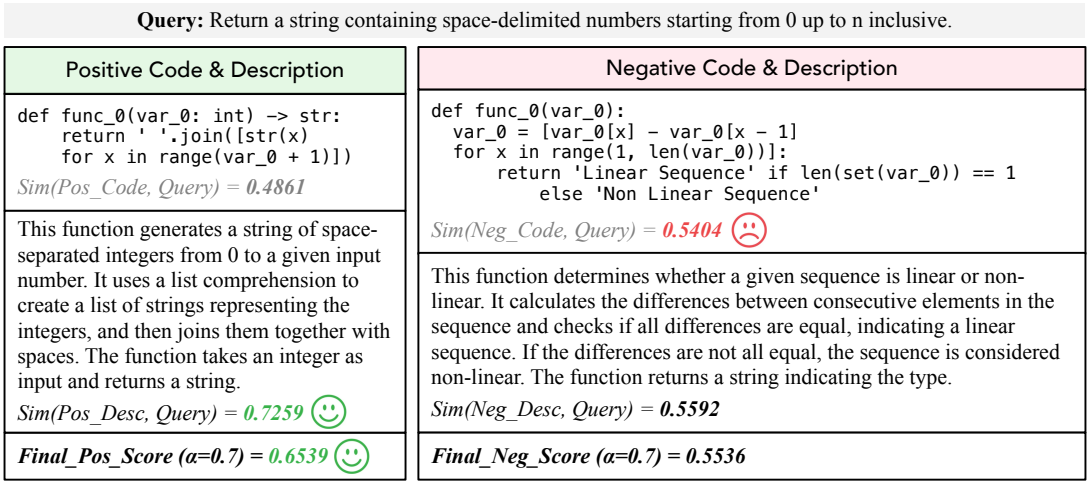


Figure 6: Case Study 1 (HumanEval/15): The baseline incorrectly ranks a function that checks for linear sequences higher based on code similarity alone, but semantic-augmented reranking correctly identifies the ground truth solution by leveraging description similarity.

Semantic-Augmented Reranking Example. As shown in Figure 6, for a query asking to generate space-separated integers, the baseline mistakenly prioritizes an unrelated function analyzing sequence linearity. In contrast, when reranking based on generated descriptions, the correct function receives a higher relevance score. The combined score in SACL still correctly prioritizes the relevant code, lifting it from rank 18 to the top. This improvement arises because the descriptions capture high-level functional intent—e.g., "generates a string of integers"—which is lost when identifiers and docstrings are normalized. The reranker can thus discriminate relevance based on semantic meaning rather than textual overlap, correcting the baseline's lexical bias.

In-Context Localization Example. Similarly, Table 9 presents two cases where our semantic-augmented localization method correctly identifies the source files for real-world GitHub issues but the Agentless baseline does not. For instance, in the first issue, the baseline misattributes the issue to the field accessor logic in ('__init__.py'). In comparison, with the file description that mentions the "Choices" class, SACL locates the correct file, "enums.py", where the faulty "enum" string representation is actually defined. Similarly, for the second issue, our method accurately links the issue to the sign function in complexes.py, rather than applying a generic patch to the core function class. We hypothesize that these successes stem from our file-level descriptions, which often include module-level purposes and inter-file relationships—semantic signals absent in raw filenames or directory trees.

5 Related Work

Code Retrieval. Code retrieval research has established key benchmarks and representation models that inform our understanding of textual bias. CodeSearchNet (Lu et al., 2021) provides foundational benchmarks for semantic code search with a corpus of 6 million functions across six programming languages, while CodeXGLUE (Husain et al., 2020) extends this with comprehensive evaluation protocols spanning 10 tasks for cross-modal retrieval between natural language and code. Building on these benchmarks, several representation models have emerged to bridge the semantic gap. CodeBERT (Feng et al., 2020) establishes foundational work on joint text-code representations using bimodal pre-training, while UniXcoder (Guo et al., 2022) advances this by incorporating structural information from Abstract Syntax Trees through contrastive learning. CodeT5+ (Wang et al., 2023b) further develops flexible encoder-decoder architectures with mixture pretraining objectives to achieve state-of-the-art performance. These advances provide essential context for understanding how current models process textual versus functional aspects of code, directly motivating our investigation of retrieval biases through systematic normalization experiments.

Retrieval-Augmented Generation for Code. Retrieval-augmented code generation (RACG) incorporates external retrieved snippets into the gen-

eration pipeline to improve performance. Early foundational work, REDCODER, (Parvez et al., 2021) introduced bi-modal text-code retrieval databases for code generation and summarization, establishing the foundation for subsequent RACG research. Recent works such as ReACC (Lu et al., 2022) and Repocoder (Zhang et al., 2023) demonstrate gains by supplying functionally relevant examples during generation. Similarly, Code-RAG Bench (Wang et al., 2025) offers a standardized benchmark for evaluating RACG systems across programming tasks, retrieval quality, and computational efficiency. Other efforts extend RACG to novel applications such as universal information extraction to generate task-specific-extractors (Guo et al., 2023). While these methods highlight the promise of RACG, they implicitly assume effective retrievers, which is not practical. Our work contributes to this area by revealing and combating a core limitation in current retrievers: a strong bias toward superficial lexical signals.

Complementary to RACG, generation-augmented retrieval (GAR) approaches tackle retrieval effectiveness from the query side. The foundational GAR framework (Mao et al., 2020) demonstrates how generated content can enhance retrieval for open-domain question answering, inspiring subsequent query expansion techniques (Wang et al., 2023a) that contrast with document-side augmentation approaches. In the code domain, ReCo (Li et al., 2024) apply GAR principles through query rewriting for code search, achieving improvements that we compare against and outperform in our experiments. Additionally, the HyDE approach (Gao et al., 2023) generates hypothetical documents for zero-shot dense retrieval, sharing conceptual similarities with semantic description generation though applied to different domains. These GAR-based methods, while effective, still face the fundamental challenge of retrieval bias that our work systematically analyzes and addresses through document-side augmentation.

LLM-Based Fault Localization. Recent work has shifted towards complex, real-world scenarios through benchmarks like SWE-Bench (Jimenez et al., 2023), featuring actual GitHub issues requiring codebase comprehension and bug-fixing. In parallel, LLM capabilities have enabled significant advances in fault localization (FL). FlexFL (Xu et al., 2025) incorporates open-source LLMs in a two-stage process to leverage bug-related information to identify and refine buggy locations. AgentFL (Qin et al., 2025) models FL as a human-like process with specialized agents for comprehension, navigation, and confirmation steps. Agentless (Xia et al., 2024) utilize a simplistic three-phase approach to narrow down fault locations from file-level locations to line-level. Unlike previous approaches focused on improving agent architectures or specialized tools, SACL tackles a fundamental limitation in code retrievers themselves: their reliance on surface-level textual features rather than functional semantics.

6 Conclusion

We conduct systematic normalization experiments and uncover two key biases for current code retrieval systems: (1) heavy dependence on textual cues over semantic understanding, and (2) consistent preference for well-documented code, even if the documentation is irrelevant. To address these issues, we propose SACL, a framework that augments code retrieval and localization with semantic information through natural language descriptions. Experiments demonstrate SACL's effectiveness in both code retrieval and code generation on three widely used benchmarks: HumanEval, MBPP, and SWE-Bench-Lite. For instance, SACL improves code retrieval by up to 12.8% Recall@1 on HumanEval and up to 8.0% file localization accuracy on SWE-Bench-Lite. These gains translate directly into better downstream code generation quality (e.g., up to 4.88% Pass@1 gain on HumanEval).

Limitations

Some of our method's limitations are: (1) We do not attempt to apply our semantic augmentation techniques to agentic methods for code generation, (2) We do not explore the performance of SACL on other Repo-level coding benchmarks, Lastly, (3) We focus primarily on function retrieval when analyzing lexical-level bias in current code retrieval techniques.

Acknowledgments

We thank Daniel Chechelnitsky, Nikhil Kandukuri, and Graham Neubig for their helpful feedback and compute provisions on this work. This work was supported in part by NSF grant DSES-2222762.

References

Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. 2018. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)*, 51(4):1–37.

Jacob Austin, Augustus Odena, Maxwell I. Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. 2021. Program synthesis with large language models. *CoRR*, abs/2108.07732.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, and 1 others. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Sunhao Dai, Yuqi Zhou, Liang Pang, Weihao Liu, Xiaolin Hu, Yong Liu, Xiao Zhang, Gang Wang, and Jun Xu. 2024. Neural retrievers are biased towards llm-generated content. In *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, KDD '24, page 526–537. ACM.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. Codebert: A pre-trained model for programming and natural languages. *Preprint*, arXiv:2002.08155.

Luyu Gao, Xueguang Ma, Jimmy Lin, and Jamie Callan. 2023. Precise zero-shot dense retrieval without relevance labels. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1762–1777.

Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. Unixcoder: Unified cross-modal pre-training for code representation. *Preprint*, arXiv:2203.03850.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. 2024. Deepseek-coder: When the large language model meets programming – the rise of code intelligence. *Preprint*, arXiv:2401.14196.

Yucan Guo, Zixuan Li, Xiaolong Jin, Yantao Liu, Yutao Zeng, Wenxuan Liu, Xiang Li, Pan Yang, Long Bai, Jiafeng Guo, and Xueqi Cheng. 2023. Retrieval-augmented code generation for universal information extraction. *Preprint*, arXiv:2311.02962.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, and 1 others. 2024. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*.

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*.

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2020. Codesearchnet challenge: Evaluating the state of semantic code search. *Preprint*, arXiv:1909.09436.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2023. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770*.

Vladimir Karpukhin, Barlas Oguz, Sewon Min, Patrick SH Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen-tau Yih. 2020. Dense passage retrieval for open-domain question answering. In *EMNLP (1)*, pages 6769–6781.

Heiko Koziolek, Sten Grüner, Rhaban Hark, Virendra Ashiwal, Sofia Linsbauer, and Nafise Eskandani. 2024. Llm-based and retrieval-augmented control code generation. In *Proceedings of the 1st International Workshop on Large Language Models for Code*, pages 22–29.

Haochen Li, Xin Zhou, and Zhiqi Shen. 2024. Rewriting the code: A simple method for large language model augmented code search. *arXiv preprint arXiv:2401.04514*.

Shuai Lu, Nan Duan, Hojae Han, Daya Guo, Seung won Hwang, and Alexey Svyatkovskiy. 2022. Reacc: A retrieval-augmented code completion framework. *Preprint*, arXiv:2203.07722.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, and 3 others. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. *Preprint*, arXiv:2102.04664.

Yuning Mao, Pengcheng He, Xiaodong Liu, Yelong Shen, Jianfeng Gao, Jiawei Han, and Weizhu Chen. 2020. Generation-augmented retrieval for open-domain question answering. *arXiv preprint arXiv:2009.08553*.

Md Rizwan Parvez, Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Retrieval augmented code generation and summarization. *arXiv preprint arXiv:2108.11601*.

Yihao Qin, Shangwen Wang, Yiling Lou, Jinhao Dong, Kaixin Wang, Xiaoling Li, and Xiaoguang Mao. 2025. Agentfl: Scaling llm-based fault localization to project-level context. *Preprint*, arXiv:2403.16362.

Aivin V Solatorio. 2024. Gistembed: Guided in-sample selection of training negatives for text embedding fine-tuning. *arXiv preprint arXiv:2402.16829*.

Nandan Thakur, Nils Reimers, Andreas Rücklé, Abhishek Srivastava, and Iryna Gurevych. 2021. Beir:

A heterogenous benchmark for zero-shot evaluation of information retrieval models. *arXiv preprint arXiv:2104.08663*.

Liang Wang, Nan Yang, and Furu Wei. 2023a. Query2doc: Query expansion with large language models. *arXiv preprint arXiv:2303.07678*.

Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi D. Q. Bui, Junnan Li, and Steven C. H. Hoi. 2023b. Codet5+: Open code large language models for code understanding and generation. *Preprint*, arXiv:2305.07922.

Zora Zhiruo Wang, Akari Asai, Xinyan Velocity Yu, Frank F. Xu, Yiqing Xie, Graham Neubig, and Daniel Fried. 2025. Coderag-bench: Can retrieval augment code generation? *Preprint*, arXiv:2406.14497.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. 2024. Agentless: Demystifying llm-based software engineering agents. *Preprint*, arXiv:2407.01489.

Chuyang Xu, Zhongxin Liu, Xiaoxue Ren, Gehao Zhang, Ming Liang, and David Lo. 2025. Flexfl: Flexible and effective fault localization with open-source large language models. *IEEE Transactions on Software Engineering*, 51(5):1455–1471.

Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. 2023. Repocoder: Repository-level code completion through iterative retrieval and generation. *Preprint*, arXiv:2303.12570.

A Appendix

A.1 Experimental Details

Datasets. We evaluate our approach on three widely-used code benchmarks: Humaneval (Chen et al., 2021) consisting of 164 hand-written programming problems with function signatures and test cases; MBPP (Austin et al., 2021) containing 974 Python programming tasks with natural language descriptions and test cases; and SWE-Bench Lite (Jimenez et al., 2023) with 300 real-world GitHub tasks representing realistic software engineering scenarios.

Implementation Details. For our code reranking approach, we use GIST-large (Solatorio, 2024) as the base embedding model for code retrieval. We employ Llama-3.1-8B-Instruct to generate concise descriptions (under 100 words) for each code snippet, focusing on functionality, algorithm, and purpose. The prompt template instructs the model to analyze the code and provide a clear description of what it does. For reranking, we combine the embedding similarity scores from both code and descriptions using Equation 1 with $\alpha = 0.7$.

As for code generation experiments, for HumanEval for MBPP, we experiment with state-of-the-art coding models: Qwen2.5-Coder-7B (Hui et al., 2024) and Deepseek-coder-7b (Guo et al., 2024). For SWE-Bench Lite, we integrate our approach with the Agentless pipeline (Xia et al., 2024), using both GPT-4o-mini and GPT-4o for both issue solving and the file summarization step in SACL. More specifically, we augment the localization steps for relevant and irrelevant file identification.

Computational costs. For Semantic-Augmented Code Reranking, we employ open-source models hosted locally to maintain cost efficiency. As indicated in our experimental results, we utilize Llama-3.1-8B-Instruct, which represents a relatively lightweight model choice. For practical deployment scenarios, code descriptions can be pre-computed and cached, effectively eliminating inference-time overhead. The one-time computational cost of description generation is substantially offset by the performance improvements achieved.

For repository-level tasks evaluated on SWE-Bench-Lite, our generated file descriptions average 65.85 completion tokens in length, with prompts averaging 2,769.22 tokens (including the target file contents). Across all SWE-Bench repositories, we process approximately 3,636 Python files, resulting in an estimated cost of $0.0148 per file description generation. This translates to an average additional cost of $0.024 per SWE-Bench example, demonstrating the cost-effectiveness of our semantic augmentation approach.

Query: Given an array with non-negative integer nodes, pluck the smallest even integer. If multiple found, return the smallest. If none found, return [].

Positive Code	Positive Code Description

`def func_0(var_0):`
`  if len(var_0) == 0:`
`  return []`
`  var_1 = list(filter(lambda x:`
`  x % 2 == 0, var_0))`
`  if var_1 == []:`
`  return []`
`  return [min(var_1),`
`  var_0.index(min(var_1))]`
 	
Code Score: 0.4713
	This function takes a list of integers as input and returns a list containing the smallest even number in the input list and its index. If the input list is empty or contains no even numbers, the function returns an empty list. The function uses list comprehension and the built-in filter function to find even numbers, and then finds the minimum and its index using the min and index methods.
	
	Description Score: 0.6383

Negative Code	**Negative Code Description**

`def func_0(self, value, list_num, index):`
`  self.value = value`
`  self.list_num = list_num`
`  self.index = index`
`def func_1(self, other):`
`  return self.value < other.value`
 	
Code Score: 0.5446
	This code implements a binary search algorithm to find the maximum subarray sum within a given list of arrays. It uses a priority queue to efficiently find the maximum and minimum values in the subarrays. The algorithm iteratively selects the subarray with the maximum sum, updates the maximum and minimum values, and repeats until the end of the subarray is reached.
	
	Description Score: 0.5057

Combined Retriever Scores (α=0.7): Positive Example: **0.5882**, Negative Example: 0.5174

Table 8: Case Study 2 (HumanEval/68): The baseline incorrectly ranks code for binary search higher than the correct solution for finding the smallest even number. Despite lower code similarity, semantic-augmented reranking correctly prioritizes the ground truth by leveraging description similarity.

Table 9: Case studies showing how semantic-augmented localization helps identify the correct files for modification which leads to correct patches being generated.

django-11964: "The value of a TextChoices / IntegerChoices field has a differing type"

Repository Structure	Localized Files	Patch Snippet

Baseline:
django/
| ...
| db/
| | ...
| | models/
| | | ...
| | | enums.py
 	
SACL method:
django/
| ...
| db/
| | ...
| | models/
| | | ...
| | | enums.py
- This Python file provides a custom implementation of enums with additional features such as ...
	Baseline:
	django/contrib/admin/options.py
	django/.../templates.py
	django/db/models/fields/__init__.py
	
	SACL method:
	django/db/models/fields/__init__.py
	django/db/models/enums.py
	django/db/models/base.py
		Baseline (wrong file):
		`# in django/db/models/fields/__init__.py`
		`- return getattr(obj, self.attname)`
		`+ value = getattr(obj, self.attname)`
		`+ if isinstance(value, enum.Enum):`
		`+  return value.value`
		`+ return value`
		SACL method (correct file):
		`# in django/db/models/enums.py`
		`class Choices(enum.Enum, metaclass=...):`
		`-  pass`
		`+  def __str__(self):`
		`+  return str(self.value)`

sympy-19487: "Rewrite the sign function in terms of Abs in SymPy"

Repository Structure	Localized Files	Patch Snippet

Baseline:
sympy/
| ...
| functions/
| | ...
| | elementary/
| | | ...
| | | complexes.py
 	
SACL method:
sympy/
| ...
| functions/
| | ...
| | elementary/
| | | ...
| | | complexes.py
- This Python file defines several mathematical functions for symbolic computation using the SymPy library ...
	Baseline:
	sympy/core/expr.py
	sympy/core/function.py
	sympy/.../miscellaneous.py
	...
	
	SACL method:
	sympy/core/function.py
	sympy/.../complexes.py
	sympy/.../miscellaneous.py
	...
		Baseline (wrong file):
		`# in sympy/core/function.py`
		`+ def _eval_Abs(self, *args, **):`
		`+  from sympy import Abs`
		`+  if len(args) == 1:`
		`+  arg = args[0]`
		`+  if arg.is_zero:`
		`+  return 0`
		`+  return arg / Abs(arg)`
		SACL method (correct file):
		`# in sympy/.../complexes.py`
		`class sign(Function):`
		`+ def evalrewrite_as_Abs(self, arg, **):`
		`+  if arg.is_zero:`
		`+  return S.NaN`
		`+  return arg / Abs(arg)`