2025 IEEE/ACM 47th International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER) | 979-8-3315-3711-1/25/$31.00 ©2025 IEEE | DOI: 10.1109/ICSE-NIER66352.2025.00025

How Propense Are Large Language Models at Producing Code Smells? A Benchmarking Study

Alejandro Velasco[1], Daniel Rodriguez-Cardenas[1], Luftar Rahman Alif[2], David N. Palacio[1], Denys Poshyvanyk[1]
[1]Department of Computer Science, William & Mary
[2]Department of Software Engineering, University of Dhaka
{svelascodimate, dhrodriguezcar}@wm.edu, bsse1120@iit.du.ac.bd, {danaderpalacio, dposhyvanyk@}wm.edu

***Abstract*—Large Language Models (*LLMs*) have shown significant potential in automating software engineering tasks, particularly in code generation. However, current evaluation benchmarks, which primarily focus on accuracy, fall short in assessing the quality of the code generated by these models, specifically their tendency to produce code smells. To address this limitation, we introduce *CodeSmellEval*, a benchmark designed to evaluate the propensity of *LLMs* for generating code smells. Our benchmark includes a novel metric: *Propensity Smelly Score (PSC)*, and a curated dataset of method-level code smells: *CodeSmellData*. To demonstrate the use of *CodeSmellEval*, we conducted a case study with two state-of-the-art *LLMs*, CodeLlama and Mistral. The results reveal that both models tend to generate code smells, such as *simplifiable-condition* and *consider-merging-isinstance*. These findings highlight the effectiveness of our benchmark in evaluating *LLMs*, providing valuable insights into their reliability and their propensity to introduce code smells in code generation tasks.**

***Index Terms*—LLMs, Smells, Benchmark, Interpretability.**

I. Introduction

Large Language Models (*LLMs*) have been used to automate multiple software engineering (SE) tasks, including code completion [1]–[3], code summarization [4], program repair [5], clone detection [6], and assertion generation [7]. LLMs' applications have expanded from classification tasks (*e.g.,* defect detection, requirements triage) to generative tasks, *i.e.,* sequence synthesis in which *LLMs* generate text or code from a given prompt [8]. Ergo, ensuring LLMs' prediction quality is crucial, particularly for SE tasks that require rigorous evaluation due to the complexity of source code. In this work, we evaluate the propensity of *LLMs* to generate or predict *code smells*.

Code smells are symptoms of poor design and implementation choices, negatively affecting code comprehensibility and maintainability [9], [10]. Recent works have explored *LLMs* to detect and refactor smells [9], [11]–[18], showing promising results when validated against ground truth. However, traditional metrics such as BLEU [19], CodeBLEU [20], ROUGE [21], and METEOR [22] fail to capture *LLMs*' *tendency* to introduce code smells at inference time. This tendency information is especially relevant for developers using commercial *LLMs* (*e.g.,* ChatGPT, Copilot, and Claude[1]), which may generate smelly snippets: `invalid-names`, `too-many-arguments`, or `unnecessary-lambda`. Although *LLMs* are largely investigated in SE life-cycle [23], little is known about the likelihood *LLMs* generate smelly code.

Despite the apparent success of using *LLMs* for automating software tasks, developers still face two key challenges: 1) they need more information to assess which *LLM* is more reliable, and 2) they cannot evaluate model performance beyond accuracy, as traditional metrics often overestimate code quality. This leads to important questions: *What is the propensity of an LLM to generate code smells?* and *Which types of code smells are most propense to be generated?*

To address these concerns, we propose a new benchmark, *CodeSmellEval*, designed to estimate the propensity of an *LLM* at generating smells. *CodeSmellEval* draws inspiration from *Syntax Decomposition* [24], [25] and introduces a model-agnostic evaluation metric called the *Propensity Smelly Score (PSC)*. *PSC* analyzes the logits from the final layer of an *LLM*, providing insights into the model's propensity of generating code smells. Additionally, our benchmark includes a new dataset, *CodeSmellData*, comprising $142k$ unique method-level code smells of 13 different types, mined from GitHub.

To demonstrate the utility of our benchmark, we designed an exploratory case study using *CodeSmellEval* in two *LLMs* (*i.e.,* CodeLlama and Mistral). We assessed the distribution of *PSC* values for each type of smell in both models. Our analysis revealed that both models are propense to generate the same types of smell, with a few exceptions. For example, *consider-merging-isinstance* (R1701), *chained-comparison* (R1716), and *broad-exception-caught* (W0718) were among the top, whereas *disallowed-name* (C0104), *too-many-arguments* (R0913), and *non-ascii-name* (C2401) were bellow the *PSC* propensity threshold. We hope that our findings will shed light on the propensity of current *LLMs* to introduce code smells, enabling a more systematic and rigorous evaluation of code quality beyond canonical accuracy.

To summarize, our key contributions are as follows: 1) a new metric to evaluate the propensity of *LLMs* to produce smells during code generation (*PSC*), 2) a new dataset *CodeSmellData* of Python methods with *Pylint* smells, 3) an exploratory case study to highlight the propensity of two current popular open-source *LLMs* to produce smells, and 4) and notebooks and code packages with instructions to replicate the experiments and use our benchmark [26].

[1] https://chatgpt.com, https://copilot.microsoft.com, https://claude.ai

2832-7632/25/$31.00 ©2025 IEEE
DOI 10.1109/ICSE-NIER66352.2025.00025

II. Benchmark

In this section, we present our benchmark, *CodeSmellEval*, which consists of three key components: (1) *PSC*, a metric designed to estimate the propensity of an *LLM* to introduce code smells, (2) our evaluation dataset, *CodeSmellData*, comprising $142k$ curated instances of method-level code smells mined from GitHub, and (3) a *protocol*, which outlines the methodology for using our benchmark.

A. Propensity Smelly Score (PSC)

PSC is an evaluation metric that works by extracting the non-normalized log-probabilities (logits) Z for each token prediction from the last hidden layer of a decoder-based transformer (*e.g.,* GPT). To estimate the probabilities of expected tokens in a sequence w, we apply the softmax function (σ) to each logit z_i. For a token $w_i = t$, where t belongs to the vocabulary V, we compute the probability $P(w_i = t|w_{<i}) \approx \sigma(z_i)_t = e^{z_{i,t}}/\sum_{j=1}^{|V|} e^{z_{i,j}}$. In this equation, $z_{i,t}$ represents the logit for the expected token t at position i, and the denominator normalizes the probabilities by summing the exponentiated logits for all vocabulary tokens. Since decoder-based models are auto-regressive, the preceding context influences the computation of $\sigma(z_i)_t$.

Meaningful Structures $\mathbb{M}$. Using an alignment function (δ), tokens $w_i \in w$ are grouped into a meaningful structure $\mu \in \mathbb{M}$ (Eq. 1). Then, an aggregation function (θ) (Eq. 2) computes a central tendency statistic (*e.g.,* median, mean, mode) of their probabilities, resulting in an overall probability estimate for predicting μ (*i.e.,* propensity score). Once $P(w_i|w_{<i})$ is estimated, the likelihood of each expected token is used to calculate the value of μ. Each token position probability is treated as independent because NTPs (Next Token Predictions) are generated auto-regressively.Eq. 2 outlines how to compute the *PSC* for a meaningful concept μ. Indexes pointing out smelly code are defined as $0 \leq i \leq k \leq j \leq |w|$.

$$\delta_\mu(w) : w \to (i,j), \mu \in \mathbb{M} \tag{1}$$

$$\theta_\mu(w,i,j) = \mathbb{E}_{k=i}^{j}[P(w_k|w_{0\ldots k-1})] \tag{2}$$

Code Smells. The definition of the set of meaningful structures $\mathbb{M}$ used in function δ (Eq. 1) depends on the problem context. For example, token probabilities can be aggregated using syntax-based decomposition, based on elements defined by the grammar of a programming language (*e.g.,* identifiers, conditionals, statements). In this paper, we propose defining the set of concepts $\mathbb{M}$ using types of smells at method-level. Table I illustrates the full taxonomy of code smells in $\mathbb{M}$ included in *CodeSmellData*.

Global Estimates. Finally, we compute the average of Eq. 2 for a given code smell type (μ) across all code snippets s in the dataset (*CodeSmellData*) to estimate the overall *PSC* of code smells generated by an *LLM*. We define a *propensity threshold* $\lambda = 0.5$ to determine whether the *LLM* is *propense* to generate the code smell, with *PSC* $\geq \lambda = 0.5$ indicating a higher propensity. We use this estimate to answer $\mathbf{RQ}_1$. The propensity threshold, based on the work of Karpaty et al. [27], is set at 0.5 to indicate that code smells are more likely to be produced than flipping a coin. In addition, Fig. 2 depicts an example of computing *PSC* for a Python snippet containing the code smells: *invalid-name* with a *PSC* of 0.6 above the propensity threshold and *comparison-of-constants* with a *PSC* of 0.206.

TABLE I: Method-level code smells instances included in *CodeSmellData*.

Type	**ID**	**Code Smell** $\mu \in \mathbb{M}$	**Count**
Convention	C0103	*invalid-name*	125815
	C0121	*singleton-comparison*	1089
	C3001	*unnecessary-lambda-assignment*	666
	C2401	*non-ascii-name*	583
	C0104	*disallowed-name*	174
Refactor	R0913	*too-many-arguments*	4738
	R1702	*too-many-nested-blocks*	1273
	R0916	*too-many-boolean-expressions*	289
	R1701	*consider-merging-isinstance*	132
	R1716	*chained-comparison*	128
Warning	W0718	*broad-exception-caught*	4384
	W0719	*broad-exception-raised*	3150
	W0108	*unnecessary-lambda*	396

*Refer to *Pylint* [28] for detailed smells' description.

B. CodeSmellData

To mitigate the risk of data leakage during the smells evaluation, we created a new testbed: *CodeSmellData* using Galeras [29]. We mined popular open-source Python repositories from GitHub, extracting a total of $232,715$ methods. The selection criteria included repositories published between January 2022 and December 2024, with a minimum of $3,500$ stars, ensuring that only well-maintained and highly rated repositories were considered. We then applied *Pylint* [28] analysis to these methods, detecting $79,574$ with at least one code smell.

Dataset Curation. From this filtered set of methods, we identified $156,151$ instances of smells across 30 distinct types. Using random sampling, two authors manually validated instances from each code smell type to confirm true positives (TP) and false positives (FP) with an 80% confidence level and 15% margin error. Discrepancies in the validation results were resolved through discussions among the authors. We found that five code smells—*inconsistent-return-statements*, *too-many-branches*, *too-many-return-statements*, *too-many-statements*, and *unbalanced-tuple-unpacking*—had near to zero precision. For *unbalanced-tuple-unpacking*, the number of FPs exceeded the number of TPs, while for the other cases, *Pylint* failed to identify the exact location on the other code smell types of the smell within the method due to parsing errors. Additionally, 12 smells were excluded because they had fewer than 100 instances, making them not representative (we observed that all *PSC* distributions tend to resemble a Gaussian distribution with at least 100 instances). Finally, we ensured that all code smell instances in *CodeSmellData* originated from unique Python methods, avoiding any repetition of source code among code smell types. The resulting dataset consists of $142,817$

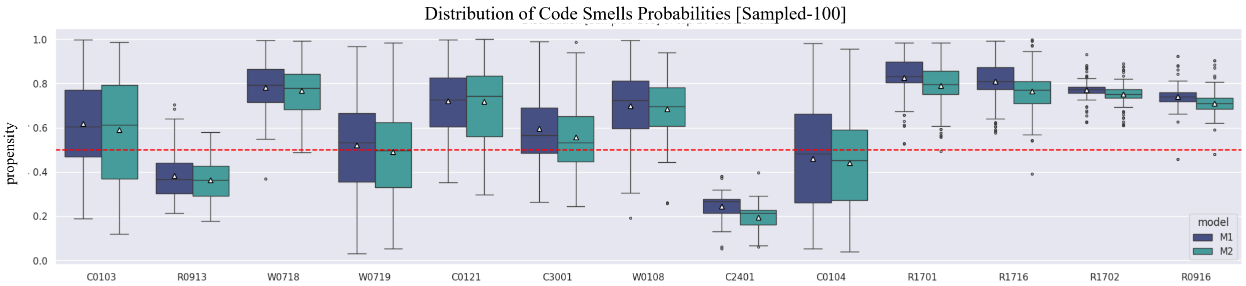

Fig. 1: Bootstrapped propensity of smells (100 instances per sample). The red line at 0.5 indicates the error threshold.

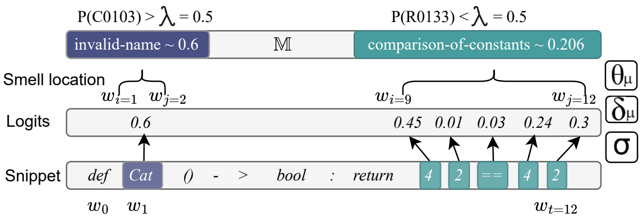

Fig. 2: Example of *PSC* computation for invalid-name (C0103) and comparison-of-constants (R0133), with the former surpassing the propensity threshold $\lambda = 0.5$.

code smell instances from 13 confirmed types, categorized into Refactoring, Warnings, and Conventions, as shown in Table I.

Code Smell Location. In our case study, we used the information provided by *Pylint* (*i.e.,* starting and ending row and column) to locate the exact portion of each code snippet containing a code smell, which allowed us to implement the alignment function δ as introduced in Eq. 1.

C. Protocol

PSC evaluates the propensity of *LLMs* to generate specific types of code structures by analyzing next-token predictions. Our benchmark computes the probability for each expected token in the sequences from *CodeSmellData*. The steps are as follows: (1) encode each sequence using the *LLM*'s tokenizer, (2) compute the logits for each token using the *LLM*'s forward method, (3) apply the softmax function (σ) to calculate the probability of each expected value, (4) group tokens corresponding to a given code smell using the alignment function (δ), and (5) compute the *PSC* for each code smell using a central tendency statistic (Eq. 2). The resulting *PSC* estimates the propensity of each smell to be generated by an *LLM*.

III. Case Study

To demonstrate the practical application of our benchmark, we conducted a case study evaluating the propensity of two *LLMs* to generate the code smells in *CodeSmellData*. We formulated the following main research question:

RQ$_1$ [Propensity of Code Smells] What types of code smells are more propense to be generated by M1 and M2?

Selected *LLMs*. Although *PSC* is model-agnostic, we selected two popular decoder-based transformers as they are well-suited for generative tasks. The first model, CodeLlama-7b-Instruct-hf (M1) [30], has a vocabulary size of $32,016$ tokens. The second model, Mistral-7B-v0.3 (M2) [31], has a vocabulary size of $37,768$ tokens. Both models have 7 billion parameters, 32 hidden layers, and 32 attention heads. The models were loaded on an Ubuntu 20.04 system with an AMD EPYC 7532 32-Core CPU, an NVIDIA A100 GPU with 40GB VRAM, and 1TB of RAM.

TABLE II: Top-5 highest and Top-3 lowest (gray bkg.) smells ranked by *PSC* (in [avg±std]).

Code Smell	M1 *PSC*	M2 *PSC*	ME - 95%
R1701	0.80 ± 0.08	0.80 ± 0.10	5%
R1716	0.80 ± 0.10	0.77 ± 0.10	5%
W0718	0.80 ± 0.12	0.77 ± 0.10	10%
R1702	0.77 ± 0.05	0.75 ± 0.06	10%
R0916	0.73 ± 0.06	0.71 ± 0.06	8%
C0104	0.46 ± 0.25	0.44 ± 0.23	7%
R0913	0.40 ± 0.13	0.36 ± 0.10	10%
C2401	0.24 ± 0.07	0.20 ± 0.06	9%

*ME - Margin Error for sample size of 100.

Evaluation Methodology. To address **RQ**$_1$, we computed *PSC* global estimates (refer to Sec. II-A) for both models (*i.e.,* M1 and M2) using the collected snippets for all identified code smells. Due to memory constraints, we limited the evaluation to datapoints in *CodeSmellData* with a maximum size of 400 tokens. We further sampled 100 datapoints from each of the 13 code smells with at least 100 instances in the dataset to ensure fair statistical treatment. Also, note that all the selected datapoints came from distinct Python methods. We then followed the protocol steps (refer to Sec. II-C) to compute the global estimates of *PSC* for each code smell.

A. Results & Discussion

Fig. 1 presents the probability scores computed for each code smell using M1 and M2. Remarkably, 10 out of 13 code smells have a probability higher than the propensity threshold of 0.5, indicating that both models are propense to generating these smells. As shown in Table II, the top five smells with the highest *PSC* in both models are *consider-merging-isinstance* (R1701), *chained-comparison* (R1716), *broad-exception-caught* (W0718), *too-many-nested-blocks* (R1702) and *too-many-boolean-expressions* (R0916). Conversely, *disallowed-name* (C0104), *too-many-arguments* (R0913) and *non-ascii-name* (C2401) are bellow the propensity threshold for both models.

Upon examining the computed distributions of *PSC* for both the highest and lowest scoring code smells in each model, as shown in Fig. 3, we observe a significant difference. The

PSC score for *consider-merging-isinstance* (R1701) is nearly double that of *disallowed-name* (C2401). We attribute this disparity to the fact that tokens associated with disallowed names have very low probabilities, as they are highly specific and closely tied to the context of the source code.

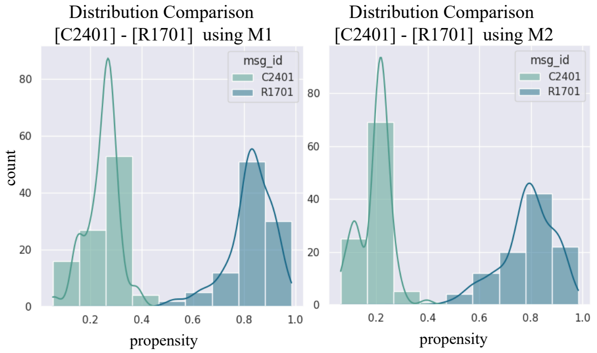

Fig. 3: Comparison of edge propensity distributions between (*consider-merging-isinstance* - R1701) and (*disallowed-name* - C2401) for both M1 and M2.

> ***RQ*$_1$ [Propensity of Code Smells]**: Both CodeLlama-7b-Instruct-hf [30] and Mistral-7B-v0.3 [31] are propense to generate code-smells Convention, Refactor, and Warnings. With few exceptions: *disallowed-name* (C0104), *too-many-arguments* (R0913), and *non-ascii-name* (C2401).

We believe that both *LLMs* are propense to generate detected code smells, as they are trained on publicly available code, which often contains quality issues [30]. Moreover, no evidence suggests that the training data of the selected *LLMs* has been curated to eliminate smells. This assumption is reinforced by our dataset creation process since we extracted smells from mined code snippets in public repositories. Nevertheless, a separate study is required to confirm this hypothesis as our goal is to demonstrate the utility of *CodeSmellEval*.

The results indicate that higher *PSC* values are correlated with a higher tendency to produce code smells, potentially reflecting models' bias toward specific types of smells. However, since *PSC* is computed from conditioned generation (*i.e.,* from code snippets containing a code smell), another study is needed to determine whether a correlation exists between *PSC* scores and the overall quality of the code generated by *LLMs*. While our observations highlight the need for larger-scale experiments and further validation, a detailed investigation into the underlying causes of code smells in generated code is beyond the scope of this paper.

IV. Related Work

Considerable research has been devoted to collecting code smell data, detecting, and repairing. Code smell often indicates deeper issues in the codebase, affecting code quality and performance [9]–[13], [32]–[34]. Our related work is focused on research on code smell datasets and papers that have reported benchmarks on generating code smells.

Nasrabadi et al. [35] introduced an SLR with the most updated code smells datasets. Most datasets are training or testing code smell detection on *LLMs* [12], [34], [36]. Datasets can be created manually [37]–[39] or automatically [40], [41] via refactorings, however, the validations are time-consuming so most automatically generated datasets are not validated [35]. Our dataset *CodeSmellData* is automatically mined and manually curated to confirm the code smell type and location.

Existing benchmarks focus on detecting code smells using different techniques (*e.g.,* SVM [36], few-shot learning [12], chain-of-thoughts [11]). CodeLMSec [12] evaluates code generation models' vulnerability to generating insecure prompts, while LCG [2], iSmell [18], and PromptSmell [42] examine *LLM*-based approaches. Unlike these, our benchmark uses logits to assess an *LLM*'s propensity for generating smelly code, rather than merely classifying or detecting it. By aligning meaningful tokens to highlight smelly segments, our approach offers a novel, interpretable metric [29], [43], suitable for black-box code generation models.

V. Future Plans

Based on the results of our case study, we demonstrated the computation of *PSC* for 13 method-level code smells. As a next step, we plan to conduct systematic experiments to evaluate the robustness of *PSC* by incorporating a wider variety of smells and a broader range of *LLMs*. This effort will involve mining more instances of underrepresented code smells in *CodeSmellData*. Specifically, we will analyze sampling error when computing *PSC* for each code smell to mitigate the risk of sampling bias and ensure the representativeness of all smell types in *CodeSmellData*. Furthermore, since our analysis has focused solely on method-level granularity, we plan to expand *CodeSmellData* to include higher-level smells, such as *god-class* and *feature-envy*. In future versions of *CodeSmellData*, we will use other static analysis tools alongside *Pylint* to reduce the dependency on a single tool for identifying smell locations, which is crucial to implement the alignment function (Eq. 1) in *CodeSmellEval*.

Our proposed benchmark identifies the types of code smells that *LLMs* are propense to generate. However, we acknowledge the need for empirical validation to demonstrate that *PSC* scores align with real-world developer expectations and impact. Future research should address how *PSC* supports practitioners in interpreting a model's behavior and how it can be effectively used in practice. For instance, *PSC* could provide actionable insights by highlighting the types of smells a model is likely to introduce, enabling developers to assess model outputs more critically and prioritize mitigation strategies. Additionally, future work should uncover the reasons behind *LLMs*' propensity to introduce specific types of smells, potentially by identifying the most relevant input features that influence the generation of these smells. This research direction is crucial for improving the trustworthiness [24] of *LLMs* and developing defense techniques to mitigate such behaviors. We believe interpretability techniques such as LIME [44] or SHAP [45] will be instrumental in achieving these goals.

References

[1] M. Ciniselli, N. Cooper *et al.*, “An empirical study on the usage of transformer models for code completion,” *IEEE TSE’22*, vol. 48, no. 12, pp. 4818–4837, 2022.

[2] F. Lin, D. J. Kim *et al.*, “When LLM-based Code Generation Meets the Software Development Process,” Mar. 2024, arXiv:2403.15852.

[3] M. White, C. Vendome *et al.*, “Toward deep learning software repositories,” in *IEEE/ACM MSR’15*, 2015, pp. 334–345.

[4] T. Ahmed and P. Devanbu, “Few-shot training llms for project-specific code-summarization,” in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE ’22. ACM, 2023.

[5] M. Jin, S. Shahriar *et al.*, “Inferfix: End-to-end program repair with llms,” in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2023. ACM, 2023, p. 1646–1656.

[6] M. White, M. Tufano *et al.*, “Deep learning code fragments for code clone detection,” in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE ’16. ACM, 2016, p. 87–98.

[7] C. Watson, M. Tufano *et al.*, “On learning meaningful assert statements for unit test cases,” in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE ’20. ACM, 2020, p. 1398–1409.

[8] C. Watson, N. Cooper *et al.*, “A systematic literature review on the use of deep learning in software engineering research,” *ACM TOSEM*, vol. 31, no. 2, Mar. 2022.

[9] M. Tufano, F. Palomba *et al.*, “When and Why Your Code Starts to Smell Bad (and Whether the Smells Go Away),” *IEEE Transactions on Software Engineering*, vol. 43, no. 11, pp. 1063–1088, Nov. 2017.

[10] F. Palomba, G. Bavota *et al.*, “On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation,” *Empirical Software Engineering*, vol. 23, no. 3, pp. 1188–1221, Jun. 2018.

[11] S. Kaniewski, D. Holstein *et al.*, “Vulnerability Handling of AI-Generated Code – Existing Solutions and Open Challenges,” Aug. 2024, arXiv:2408.08549.

[12] H. Hajipour, K. Hassler *et al.*, “CodeLMSec benchmark: Systematically evaluating and finding security vulnerabilities in black-box code language models,” in *2024 IEEE Conference on Secure and Trustworthy Machine Learning (SaTML)*. IEEE, pp. 684–709.

[13] M. L. Siddiq, J. C. S. Santos *et al.*, “SALLM: Security Assessment of Generated Code,” Sep. 2024, arXiv:2311.00889.

[14] Y. Yang, X. Zhou *et al.*, “DLAP: A Deep Learning Augmented Large Language Model Prompting Framework for Software Vulnerability Detection,” May 2024, arXiv:2405.01202.

[15] K. Lucas, R. Gheyi *et al.*, “Evaluating Large Language Models in Detecting Test Smells,” Jul. 2024, arXiv:2407.19261.

[16] D. Mahalakshmi, P. Kasinathan *et al.*, “Code smell detection using hybrid machine learning algorithms,” in *2023 5th International Conference on Inventive Research in Computing Applications (ICIRCA)*, 2023, pp. 633–638.

[17] Y. Li and X. Zhang, “Multi-Label Code Smell Detection with Hybrid Model based on Deep Learning,” in *Proceedings of the 33rd International Conference on Software Engineering and Knowledge Engineering*, Jul. 2022, pp. 42–47.

[18] Di Wu, Fangwen Mu *et al.*, “iSMELL: Assembling LLMs with Expert Toolsets for Code Smell Detection and Refactoring,” in *Proceedings of the 40th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE ’24, Sacramento, California, United States, Oct. 2024.

[19] K. Papineni, S. Roukos *et al.*, “BLEU: a method for automatic evaluation of machine translation,” in *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, ser. ACL ’02. USA: Association for Computational Linguistics, Jul. 2002, pp. 311–318.

[20] S. Ren, D. Guo *et al.*, “CodeBLEU: a Method for Automatic Evaluation of Code Synthesis,” Sep. 2020, arXiv:2009.10297.

[21] C.-Y. Lin, “ROUGE: A Package for Automatic Evaluation of Summaries,” in *Text Summarization Branches Out*. Barcelona, Spain: Association for Computational Linguistics, Jul. 2004, pp. 74–81.

[22] A. Lavie and A. 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*, ser. StatMT ’07. Association for Computational Linguistics, 2007, p. 228–231.

[23] C. Watson, N. Cooper *et al.*, “A systematic literature review on the use of deep learning in software engineering research,” *ACM Trans. Softw. Eng. Methodol.*, vol. 31, no. 2, Mar. 2022.

[24] D. N. Palacio, D. Rodriguez-Cardenas *et al.*, “Towards More Trustworthy and Interpretable LLMs for Code through Syntax-Grounded Explanations,” Jul. 2024, arXiv:2407.08983.

[25] A. Velasco, D. N. Palacio *et al.*, “Which syntactic capabilities are statistically learned by masked language models for code?” in *Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*, ser. ICSE-NIER’24. ACM, 2024, p. 72–76.

[26] WM-SEMERU, “Codesmells,” https://github.com/WM-SEMERU/CodeSmells, 2024, accessed: 2024-12-25.

[27] A. Karpathy, J. Johnson *et al.*, “Visualizing and Understanding Recurrent Networks,” Nov. 2015, arXiv:1506.02078.

[28] PyCQA, “Pylint,” Python Code Quality Authority (PyCQA), 2003.

[29] D. Rodriguez-Cardenas, D. N. Palacio *et al.*, “Benchmarking causal study to interpret large language models for source code,” in *ICSME’23*, 2023, pp. 329–334.

[30] B. Rozière, J. Gehring *et al.*, “Code Llama: Open Foundation Models for Code,” Jan. 2024, arXiv:2308.12950 [cs].

[31] A. Q. Jiang, A. Sablayrolles *et al.*, “Mistral 7B,” Oct. 2023, arXiv:2310.06825 [cs].

[32] D. Mahalakshmi, P. Kasinathan *et al.*, “Code smell detection using hybrid machine learning algorithms,” in *2023 5th International Conference on Inventive Research in Computing Applications (ICIRCA)*. IEEE, pp. 633–638.

[33] M. Siksna, I. Berzina *et al.*, “Machine learning powered code smell detection as a business improvement tool,” in *2023 IEEE 64th International Scientific Conference on Information Technology and Management Science of Riga Technical University (ITMS)*. IEEE, pp. 1–6.

[34] A. Mohsin, H. Janicke *et al.*, “Can We Trust Large Language Models Generated Code? A Framework for In-Context Learning, Security Patterns, and Code Evaluations Across Diverse LLMs,” Jun. 2024, arXiv:2406.12513.

[35] M. Zakeri-Nasrabadi, S. Parsa *et al.*, “A Systematic Literature Review on the Code Smells Datasets and Validation Mechanisms,” *ACM Computing Surveys*, vol. 55, no. 13s, pp. 1–48, Dec. 2023.

[36] B. Nguyen Thanh, M. Nguyen N. H. *et al.*, “ml-Codesmell: A code smell prediction dataset for machine learning approaches,” in *The 11th International Symposium on Information and Communication Technology*. ACM, Dec. 2022, pp. 368–374.

[37] L. Madeyski and T. Lewowski, “Mlcq: Industry-relevant code smell data set,” *Proceedings of the 24th International Conference on Evaluation and Assessment in Software Engineering*, 2020.

[38] M. Hozano, N. Antunes *et al.*, “Evaluating the accuracy of machine learning algorithms on detecting code smells for different developers,” in *International Conference on Enterprise Information Systems*, 2017.

[39] H. Nandani, M. Saad *et al.*, “DACOS—a manually annotated dataset of code smells,” in *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*. IEEE, pp. 446–450.

[40] V. Lenarduzzi, N. Saarimäki *et al.*, “The Technical Debt Dataset,” in *Proceedings of the Fifteenth International Conference on Predictive Models and Data Analytics in Software Engineering*. ACM, Sep. 2019, pp. 2–11.

[41] Y. Wang, S. Hu *et al.*, “Using code evolution information to improve the quality of labels in code smell datasets,” in *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, vol. 01, 2018, pp. 48–53.

[42] H. Liu, Y. Zhang *et al.*, “Prompt Learning for Multi-Label Code Smell Detection: A Promising Approach,” Feb. 2024, arXiv:2402.10398.

[43] D. Nader Palacio, A. Velasco *et al.*, “Toward a theory of causation for interpreting neural code models,” *IEEE Transactions on Software Engineering*, vol. 50, no. 5, pp. 1215–1243, 2024.

[44] M. T. Ribeiro, S. Singh *et al.*, “”why should i trust you?”: Explaining the predictions of any classifier,” in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD ’16. ACM, 2016, p. 1135–1144.

[45] S. M. Lundberg and S.-I. Lee, “A unified approach to interpreting model predictions,” in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, ser. NIPS’17. Curran Associates Inc., 2017, p. 4768–4777.