Code Smell Classification in Python: Are Small Language Models Up to the Task?

Igor Soares de Oliveira
Pontifícia Universidade Católica do Rio de Janeiro
Rio de Janeiro, Brazil

Joanne Carneiro
Pontifícia Universidade Católica do Rio de Janeiro
Rio de Janeiro, Brazil

Jessica Barbara da Silva Ribas
Pontifícia Universidade Católica do Rio de Janeiro
Rio de Janeiro, Brazil

Juliana Alves Pereira
Pontifícia Universidade Católica do Rio de Janeiro
Rio de Janeiro, Brazil

Abstract

Code quality is essential for maintainable and evolvable software systems. Traditional code smell detection tools rely on AST-based techniques and metric-driven heuristics, which, while effective, often lack interpretability and require specialized knowledge. This paper investigates the use of Small Language Models (SLMs) for classifying two widely studied code smells—Long Method and Long Parameter List—in Python codebases. Unlike Large Language Models (LLMs), SLMs offer lower latency, reduced computational cost, and greater privacy, making them suitable for deployment in resource-constrained environments. We systematically compare the performance of SLMs with traditional Machine Learning (ML) and Deep Learning (DL) models, and the AST-based DPy tool. Our evaluation considers precision, recall, processing time, and explainability, using a custom event-driven dataset designed for code classification. We also analyze the role of prompt engineering techniques—zero-shot and chain-of-thought—in enhancing LLM performance. Results show that SLMs achieve competitive accuracy with improved interpretability. Additionally, we release an annotated dataset comprising classifications from all approaches. This work provides new insights into lightweight, explainable, and practical methods for automated code quality assessment, and supports the broader adoption of SLMs in software engineering.

1 Introduction

Ensuring high code quality is essential for the long-term success of any software project. Fowler et al. [15] describe the term code smell as symptoms in code that, while not strictly erroneous, signal deeper design or structural problems. Code smells accumulate technical debt, increase the risk of defects, and slow down development teams [33]. Refactoring—systematic code restructuring without changing external behavior—is the primary remedy, but it relies on accurate, timely detection of such smells.

Two pervasive and impactful code smells are Long Method (LM) and Long Parameter List (LPL). A LM bundles multiple responsibilities into a single routine, making it difficult to comprehend, test, and reuse; refactoring by Extract Function or moving logic into well-named helper methods typically restores clarity and modularity [15]. A LPL burdens callers and maintainers with a proliferation of arguments—often interdependent or referring to the same conceptual entity—thereby increasing cognitive load; techniques like Introduce Parameter Object or Preserve Whole Object can collapse related parameters into coherent abstractions [6, 15]. Detecting these smells early is critical: manual inspection does not scale, and traditional metric-based tools, though precise, often lack context-sensitive explanations and struggle to adapt to evolving codebases.

Recent advances in language modeling offer a promising alternative. Small Language Models (SLMs)—defined here as pre-trained neural models under 8 billion parameters—bridge the gap between heavyweight Large Language Models (LLMs) and classic static analysis tools. With optimized architectures and curated data, SLMs can deliver performance on par with LLMs while operating with lower inference latency, reduced hardware requirements, and improved on-device privacy guarantees [38, 41]. Unlike AST-based detectors, SLMs can generate natural-language explanations alongside binary smell classifications, augmenting interpretability and aiding developers in understanding the root causes of detected issues.

In this work, we investigate whether SLMs are "up to the task" of identifying LM and LPL smells in Python code. Our contributions are fourfold: (1) We design and implement an event-driven API that integrates multiple SLMs for on-demand code smell classification, embedding prompt-engineering techniques—zero-shot and Chain-of-Thought—to elicit precise and reasoned model outputs. (2) We benchmark SLM performance against both AST-metric tools (e.g., the DPy analyzer [8]) and traditional machine-learning (e.g., Decision Trees, Random Forests, Gradient Boosting) and deep-learning models (e.g., MLP, LSTM, GRU, AutoKeras). (3) We build a large, labeled dataset of Python functions annotated for LM and LPL smells containing the classifications produced by each approach (SLMs, ML, DL, DPy and PySmell), with 230,829 records, enabling experiment replication and supporting future comparative research in the field. We also make available multilabel ML and DL models for code smell classification. (4) Implementation of a web-based frontend application aimed at providing users with an intuitive interface for submitting and classifying Python code snippets.

We demonstrate that SLMs achieve competitive classification metrics, with F1-scores up to 0.87 for LM and 0.59 for LPL using the Mistral model with Chain-of-Thought prompts. Moreover, it also provides actionable explanations that lower the barrier to refactoring for developers relatively new to code-quality concepts.

Moreover, our results show that while classical AST-based approaches still hold an edge in raw accuracy (e.g., DPY's F1-score of 0.97 for LM), SLMs offer a compelling balance between detection performance, interpretability, and operational efficiency. Moreover, prompt engineering emerges as a key factor: Chain-of-Thought strategies consistently outperform zero-shot baselines by guiding models through multi-step reasoning about code structure. Finally,

we release our dataset and evaluation framework to support reproducibility and stimulate further research into hybrid static–dynamic analysis techniques for sustainable software development.

2 Related Work

In the literature, several studies address the topic of code quality, with a predominant focus on languages such as Java and C, C++, C#. Some of these studies employ tools like inFusion [13, 31], JDeodorant [13, 31, 39] , PMD [13, 31] , and JSpIRIT [13, 31, 40], which can be integrated into IDEs to support code analysis. Additionally, AST-based techniques for code smell detection in Python have been explored, such as the PySmell [9] and DPY [8] tools. Furthemore, several studies [20, 34, 48] use ML and DL techniques to code smell classification. However, despite their effectiveness, these approaches generally require a significant level of technical expertise to be successfully applied. Tools that integrate with development environments (IDEs) often demand complex configuration procedures and in-depth familiarity with the specific environment. Similarly, ML and DL approaches typically require access to labeled datasets, model training, hyperparameter optimization, and rigorous validation processes to ensure acceptable performance. These requirements can pose substantial barriers to broader adoption, particularly among professionals without specialized knowledge in ML or those who are at the early stages of their careers in software development.

LLMs have recently gained attention as a promising tool for detecting code smells. A notable contribution in this area is the empirical study by Silva et al. [37]. This study investigates the use of ChatGPT to classify four common types of code smells (Blob, Data Class, Feature Envy, and Long Method) in Java programs. The authors evaluated the model's performance using both generic and specifically crafted prompts, finding that detailed prompts significantly improved detection accuracy. The study also revealed that ChatGPT is particularly effective in identifying smells with higher severity levels (e.g., Critical), achieving an F-measure of up to 0.52. These findings not only reinforce the potential of LLMs in software quality analysis but also provide us with valuable methodological insights into prompt design for code analysis tasks.

Wu et al. [44], proposed iSMELL, an approach that combines LLMs with specialized tools for code smell detection and refactoring in Java. iSMELL employs a Mixture of Experts (MoE) architecture to dynamically select the most appropriate set of tools for identifying different types of code smells, such as Refused Bequest, God Class, and Feature Envy. Empirical evaluation demonstrated that iSMELL achieved an average increase of 35.05% in F1-score for smell detection tasks in comparison to standalone expert tools. The study highlights that integrating specialized tools with LLMs offers a scalable and effective approach to maintaining code quality.

In contrast to previous studies, this work proposes a distinct approach to the task of code smell classification by employing two prompting strategies — zero-shot and CoT — rather than using a generic prompt, as observed in related research. This methodological choice enables a more controlled and interpretable analysis of model behavior. Furthermore, while most existing studies focus on languages such as Java, C, C++, and C#, as previously noted, this study targets the Python language, expanding the scope of applicability and contributing to a more diverse set of contexts explored in the literature. In addition, this work adopts the use of SLMs, which are more accessible and cost-effective than LLMs, thereby facilitating their adoption in academic settings, industrial environments, and by users with limited computational resources. The proposed architecture, structured around modular microservices, was designed to support both the replication of experiments and the extension of future research. Finally, we explore aspects of model explainability, including automated refactoring suggestions for the classified code samples, thus contributing to improvements in software quality and in the development of practitioners' competencies related to code quality — as evidenced by the provided artifacts.

3 Study Design

This section presents our study design, structured as presented in Figure 1 and detailed as follows.

3.1 Code Extraction and Data Processing

In this phase, we used the PySmell dataset [1]. It provides key properties that enable the extraction of source code from projects, such as NLTK[2], Boto[3], Tornado[4], Ansible[5], Matplotlib[6], NumPy[7], SciPy[8], and Django[9]. We used the fields `subject`, `tag`, and `file` to identify the corresponding repositories, determine the specific version of each project, and locate the files that were manually annotated by experts. After identifying the official GitHub repositories, an extraction algorithm was developed to retrieve the methods and classes labeled in the PySmell dataset. This algorithm relies on attributes such as `line` and `file` to ensure that the extracted code precisely matches the method or class starting at the specified line. Once the correct file and line number are located, the code segment is extracted into a dedicated folder, and the resulting file is named following the pattern `line_filename_method/class_name`. Additionally, a new CSV file is generated containing only the code elements that were successfully retrieved from the original repositories (Figure 1 – Phase 1). This ensures that only code samples manually labeled in PySmell and reliably matched in the source projects were considered in this study, while unmatched entries were excluded (see [3] for more information). Due to compatibility constraints between the PySmell dataset and the DPY tool, this study focused exclusively on extracting the LM and LPL smells. These smells were selected because they are present in both resources, thereby enabling consistent comparisons across different approaches.

3.2 Use of DPY

DPy [10] is a code smell detection tool specifically designed for Python projects and design-level code smells using heuristics based

[1] https://github.com/chenzhifei731/Pysmell
[2] https://github.com/nltk/nltk
[3] https://github.com/boto/boto
[4] https://github.com/tornadoweb/tornado
[5] https://github.com/ansible/ansible
[6] https://github.com/matplotlib/matplotlib
[7] https://github.com/numpy/numpy
[8] https://github.com/scipy/scipy
[9] https://github.com/django/django

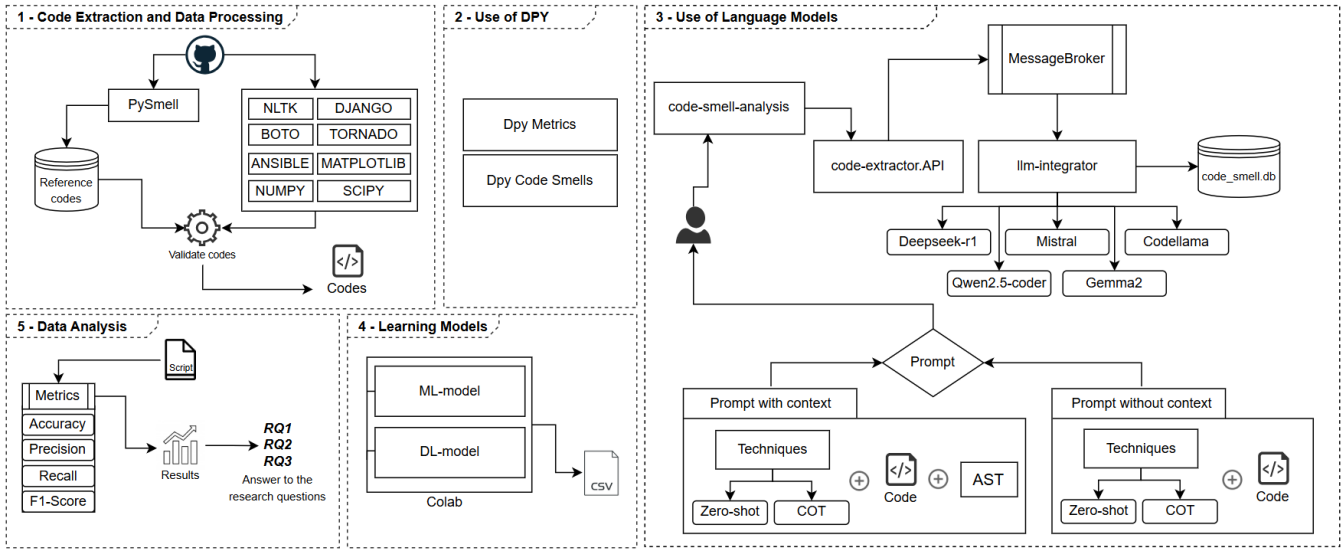

Figure 1: Code Smell Classification System Based on LMs

on metrics extracted from the Abstract Syntax Tree (AST). Although not open source, it is freely usable and allows results to be exported in CSV or JSON formats (Figure 1 – Phase 2).

The tool computes object-oriented metrics—such as LOC, NOM, NOPM, and CC—that assess structural aspects like size, complexity, and cohesion. In this study, we use these metrics both to support ML/DL classification and to inform prompt engineering for SLMs.

DPy detects 19 code smells, including the LM and LPL smells analyzed in this work. These are identified through threshold-based rules applied to code metrics. We adopt DPy as a baseline to compare its performance with ML, DL, and SLM-based methods (Section 4.1).

3.3 Use of Language Models

In this study, the architecture was developed following microservice and event-driven standards, as illustrated in Figure 1 (Phase 3 - SLMs). This architectural style leverages events to trigger actions and enable communication between decoupled services [11], promoting modularity and scalability. It involves three core components [23]: producers, routers, and consumers—where events are emitted, routed, and processed. This design allows independent deployment and evolution of services, supporting high availability and real-time processing, while facilitating maintainability and system extensibility over time.

Furthermore, the adoption of a microservice-based architecture was motivated by the need for flexibility and scalability. This model allows new types of code smells to be added incrementally, with minimal impact on existing services. As a result, the system can be progressively evolved to support different categories of smells while preserving modularity and facilitating maintenance and extensibility of the solution.

The *code-smell-analysis* application serves as a front-end interface for code submission, prompt configuration, SLM selection, and extraction type definition. It guides the user throughout the process, enhancing transparency and usability.

Operating within an anti-corruption layer (ACL) [36], the *code-extractor.api* acts as an event producer, receiving the contract[10] from *code-smell-analysis*, extracting structural elements (classes/methods) via AST [14], and dispatching events to the router in accordance with the output contract [2].

The *lm-integrator* application operates as an event consumer, processing messages using SLMs integrated via Ollama [30] or RESTful APIs [19, 28]. For this study, five SLMs were evaluated: Deepseek-R1, Qwen2.5-coder, Gemma2, Mistral, and codellama (Table 1). Results are persisted in a relational database to support further analysis and reproducibility.

Table 1: Overview of the SLMs evaluated in this work

Model Name	Parameters	Provider	Context	Quantization	Tags	Model Size
Deepseek-R1	1.5 billion	Ollama	128K	Q4_K_M	a42b25d8c10a	1.1 GB
Qwen2.5-coder	1.5 billion	Ollama	32K	Q4_K_M	6d3abb8d2d53	986 MB
Gemma2	2.61 billion	Ollama	8K	Q4_0	8ccf136fdd52	1.6 GB
Mistral	7.25 billion	Ollama	32K	Q4_0	f974a74358d6	4.1 GB
codellama	6.74 billion	Ollama	16K	Q4_0	8fdf8f752f6e	3.8 GB

3.4 Learning Models

Learning models were employed to assess whether their performance could achieve results equal to or superior to those obtained by SLMs (Figure 1 – Phase 4). Accordingly, the models were trained using Dpy metrics to evaluate the predictive performance of different ML and DL algorithms in detecting code smells. To train the models, a multilabel classification approach was adopted as the target strategy.

3.4.1 Code Classification with ML Models. ML enhances the effectiveness of code smell classification by leveraging project-related metrics [4, 25, 46]. ML models are capable of simultaneously analyzing multiple metrics and detecting correlations among them,

[10] See supplementary material [1]

allowing for the identification of real patterns associated with code smells. To train and execute the ML models, we employed the PyCaret library[11]. For this study, we selected 15 machine learning algorithms, namely: *k-Nearest Neighbor* (kNN) [24], *Linear Regression* (LR) [24], *Support Vector Machine* (SVM) [24], *Decision Tree* (DT) [21], *Random Forest* (RF) [24], *Extreme Gradient Boosting package* (XGB) [24], *Ridge Classifier* (RC) [22], *Light Gradient Boosting* (LGB) [12], *Gradient Boosting* (GB) [17], *Ada Boost* (ADA) [16], *Extra Trees* (ET) [18], *Naive Bayes* (NB) [24], *Linear Discriminant Analysis* (LDA) [5], *Quadratic Discriminant Analysis* (QDA) [32] e *Dummy Classifier* (DC) [43]. These algorithms were chosen based on a list of the most frequently used ML algorithms in defect and code smell classification fields [35].

3.4.2 Code Classification with DL Models. DL has been extensively used [27] in code smell detection, primarily due to its ability to analyze large volumes of data and identify complex patterns that traditional ML methods are unable to detect. In our study, the DL algorithms employed are: *Multi-Layer Perceptron* (MLP) [29], *Long Short-Term Memory* (LSTM) [29], *Gated Recurrent Unit* (GRU) [47], and *Autokeras*[12].

3.5 Prompt Design

Aiming to analyze the impact of incorporating syntactic information on the performance of language models, this study proposes two variations for each prompting strategy (zero-shot and CoT) [26, 42]. The selection of these approaches is based on the assumption that zero-shot enables the evaluation of the model's ability to respond without prior conditioning, whereas CoT allows for more descriptive and reasoning-driven responses. For each strategy, two variations are proposed: with and without the use of metrics extracted from the AST. This methodological distinction enables the evaluation of how the inclusion of explicit structural data influences the accuracy and robustness of code smell identification.

In the prompt *without context* configuration, prompts were constructed without the inclusion of structural information derived from the AST. The model receives only a concise instruction requesting the classification of a code snippet based on the presence of a specific code smell. This approach aims to evaluate the intrinsic ability of the language model to interpret and identify undesirable patterns solely from the textual representation of the source code. The prompt structure in this variation follows the format: textual instruction + code snippet + output formatted in JSON with the corresponding code smell.

Conversely, in the prompt *with context* variation, prompts were enriched with structural metrics extracted from the Abstract Syntax Tree (AST), providing the model with additional syntactic information that may support the inferential process. The inclusion of this structural context aims to examine the extent to which the explicit availability of code characteristics influences the accuracy and effectiveness of automated code smell classification. The prompt structure in this configuration follows the format: textual instruction + code snippet + AST metrics + output formatted in JSON with the corresponding code smell.

[11]https://pycaret.readthedocs.io/en/latest
[12]https://autokeras.com/tutorial/multi/

Zero-shot

Prompt:
Classify the code as smelly or non-smelly
Code: <add code>
[Metric: <add AST metric>] (only used for Prompt with context)
return the result as one of the following JSONs: {"smell_type":"smelly"} OR {"smell_type":"non-smelly"} AND {"explanation":"your explication about this code"}

Chain of Thought - LM

Prompt:
LMs make code difficult to understand, test, and maintain. When a method grows too large, it can exhibit high complexity, excessive conditional statements, nested loops, and multiple responsibilities, violating the Single Responsibility Principle (SRP). Additionally, LMs often mix different levels of abstraction, making the code harder to read and reuse. Changes in this type of code are more prone to errors, affecting multiple parts of the system. Based on these characteristics, the provided code may show signs of this code smell if it has too many lines, a high level of coupling, excessive parameters or local variables, and performs multiple operations that could be extracted into smaller, more cohesive functions. Does the provided code exhibit signs of a LM?
Code: <add code>
[Metric: <add AST metric>] (only used for Prompt with context)
Let's think step by step.
return the result as one of the following JSONs: {"smell_type":"smelly"} OR {"smell_type":"non-smelly"} AND {"explanation":"your explication about this code"}

3.6 Data Analysis

To conduct the evaluation of the model responses, we define three research questions (see Section 3.6.1). We employed scripts developed in Python, responsible for extracting the results stored in the code_smells.db database (Figure 1 – Phase 3) and converting them into CSV files. These files were properly organized to enable the quantitative analysis of model performance, using standard evaluation metrics: precision, recall, and F1-score. The analysis aimed to measure the models' ability to correctly classify code snippets (Figure 1 – Phase 5). Additionally, the data were organized to facilitate the replication of the study and to support future investigations using the labeled code snippets.

3.6.1 Research Questions. This study is guided by three research questions (RQs), as follows:

RQ$_1$ **What is the performance of SLMs in classifying Python code smells?**

This research question investigates the effectiveness of SLMs in detecting code smells using a baseline prompt engineering technique. To set a reference point for evaluating, we start by establishing a prompt baseline using zero-shot (see Section 3.5). We used five open-source SLMs: Deepseek-R1, Qwen2.5-coder, Gemma2, Mistral and codllama. The goal is to evaluate whether they can reliably identify smelly code fragments and classify them appropriately without prior fine-tuning. As our ground truth, we used the manual classification labels provided by the PySmell dataset [45] to determine how well SLMs align with expert judgments in real-world codebases.

RQ$_2$ **To what extent do other prompt engineering techniques differ from our baseline for executing the same tasks, in terms of performance?**

Building on our baseline, we start comparing other prompt engineering techniques. Among these comparisons, we explore whether SLMs, when guided by more refined prompting strategies, can match or surpass the effectiveness of traditional static analysis tools that rely on abstract syntax tree (AST) metrics. For that purpose, we contrast the best-performing SLM (from our baseline) with a well-established metric-based tool, DPY [7], commonly used for code smell detection. This helps clarify both the advantages and limitations of SLM-based approaches when considered as alternatives to conventional metric-based techniques. By combining these dimensions, our analysis provides actionable insights into how different prompt engineering techniques impact the overall performance and explainability of SLMs in software analysis tasks.

RQ$_3$ **What is the performance of SLMs in classifying Python code smells compared to traditional ML and DL approaches?**

This research question investigates whether SLMs can match or surpass the performance of traditional ML and DL approaches in classifying Python code smells. By comparing these paradigms, we aim to understand the trade-offs in accuracy, generalizability, and effort required for training and deploying models for automated code quality assessment.

4 Results

We present and discuss the results related to our RQs as follows.

4.1 Baseline Evaluation of Zero-Shot (RQ1)

An empirical evaluation was conducted using the prompt engineering technique in a zero-shot scenario as the baseline, with the objective of analyzing the capability of SLMs in the automatic classification of code smells in Python.

In the LM classification task, the results presented in Table 2 show that the performance of the evaluated models varied significantly. Among the five open-source LLMs evaluated, Mistral achieved the best performance, with an F1-score of 68%, precision of 83%, and recall of 68%, indicating a high capacity to correctly identify code snippets both with and without the smell. The Codellama model also showed competitive performance, with an F1-score of 57%, followed by Qwen2.5-coder with 51%. Conversely, Gemma2 obtained the lowest F1-score (31%), evidencing significant limitations in its effectiveness for this task.

Similarly, in the LPL classification task, the results presented in Table 2 reveal notable differences in performance among the models. Mistral once again stood out, achieving the highest F1-score (59%), with precision of 57% and recall of 65%, indicating consistent capability in identifying methods with excessive parameter lists. Subsequently, Codellama demonstrated competitive performance with an F1-score of 46, while Deepseek-r1 obtained 48%. On the other hand, Gemma2 exhibited a considerably lower F1-score (10%), suggesting significant limitations in detecting this type of code.

Overall, the results indicate that some LLMs, especially Mistral, are capable of reliably detecting LM and LPL code smells in Python, even without the need for fine-tuning or additional contextual information.

4.2 Impact of Prompt Engineering Techniques on SLM Performance (RQ2)

To address the research question, we conducted a comparative analysis of prompt engineering techniques—zero-shot and CoT with and without context. Table 2 shows performance varied across techniques for both LM and LPL classification.

In LM classification, Mistral consistently outperformed other models, with F1-score increasing from 0.68 zero-shot to 87% CoT and 85% CoT with context, evidencing the value of reasoning chains. Context alone (AST metrics) yielded no gain—remaining at 47%. For LPL, CoT led to slight improvements 59% to 53% and 51%, but overall gains were modest. Some models, like Gemma2, remained ineffective regardless of technique. These results indicate CoT enhances classification performance, while AST-derived context alone does not consistently help—likely due to integration limitations.

Though effective, CoT introduces computational overhead due to prompt complexity. Still, SLMs proved useful in explainability, identifying design violations (e.g., SRP), justifying predictions, and suggesting refactorings like method decomposition and parameter reduction to improve cohesion.

When compared to DPY (see Table 3), Mistral outperformed in LPL, while DPY led in LM. However, DPY lacks interpretability and shares SLMs' high cost—key considerations when choosing between SLM-based and metric-driven approaches.

In summary, refined prompting directly influences SLM performance in software analysis, supporting practical use and future research.

4.3 SLMs versus ML and DL Models (RQ3)

Table 2 shows the results relating to the performance of the SLMs compared to the traditional ML and DL models. When analyzing the LM smell, the DL Autok model stands out from the others because all of its metric values reached 100%. On the other hand, ML models such as GBC, ADA and LGB, had very low F1-score values.

When we analyzed the LPL smell, we noticed that Autok also had the best results, both among the learning models and in relation to SLM. In relation to Mistral, compared to LM and the previous smells, the values of Mistral's performance metrics dropped considerably, showing 59% in F1-score. On the other hand, ML models, such as: GBC, NB and QDA had a significant increase in performance with 92% F1-score.

The results demonstrate that DL and ML models exhibit competitive performance in the task of classifying *code smells* in Python when compared to SLMs. Among the DL-based approaches, AutoK obtained a perfect F1-score (1.00) for both *code smell* classes. However, the other DL models considered MLP and CNN did not perform satisfactorily, with F1-scores of zero across all metrics. Moreover, traditional ML models such as GBC, ADA, and LGB achieved substantially lower F1-scores than Mistral.

Overall, the findings suggest that SLMs, such as Mistral, represent a promising alternative to traditional ML and DL techniques for automated *code smell* classification, offering an advantageous balance between predictive performance, inference efficiency, and ease of deployment. Moreover, the potential of prompt-based approaches is highlighted, as they eliminate the need for costly training and

Table 2: Performance comparison of LLMs using different prompt engineering techniques for code smell classification

Class	Model	Zero-shot (Baseline)			Zero-shot with context			Chain of Thought (CoT)			CoT with context		
		Precision	Recall	F1-score	Precision	Recall	F1-score	Precision	Recall	F1-score	Precision	Recall	F1-score
LM	Deepseek-r1	0.47	0.48	0.47	0.49	0.49	0.49	0.42	0.45	0.36	0.21	0.38	0.23
	Qwen2.5-coder	0.51	0.51	0.51	0.53	0.53	0.52	0.53	0.53	0.52	0.35	0.45	0.28
	Gemma2	0.41	0.47	0.31	0.39	0.44	0.31	0.16	0.43	0.24	0.15	0.37	0.21
	Mistral	**0.83**	**0.68**	**0.68**	0.83	0.54	0.47	**0.90**	**0.85**	**0.87**	**0.86**	**0.84**	**0.85**
	codellama	0.58	0.57	0.57	**0.64**	**0.64**	**0.64**	0.48	0.48	0.40	0.52	0.51	0.43
LPL	Deepseek-r1	0.50	0.48	0.48	0.50	0.50	0.46	0.50	0.46	0.19	0.50	0.51	0.18
	Qwen2.5-coder	0.50	0.51	0.41	0.50	0.52	0.39	0.50	0.51	0.15	0.51	0.51	0.10
	Gemma2	0.51	0.54	0.10	0.51	0.54	0.10	0.51	0.50	0.02	0.51	0.50	0.02
	Mistral	**0.57**	**0.65**	**0.59**	**0.67**	**0.53**	**0.55**	**0.53**	**0.68**	**0.53**	**0.52**	**0.66**	**0.51**
	codellama	0.50	0.54	0.46	0.50	0.55	0.42	0.51	0.53	0.10	0.51	0.54	0.18

hyperparameter tuning processes, making LLMs particularly attractive for practical applications in *code smell* classification in Python.

Table 3: Comparison of LLM, DPY, ML, and DL models per code smell class, including execution time

Model	Class	Accuracy	Precision	Recall	F1-score	Exec. Time (s)
Best-SLM (Mistral)						
Mistral	LM	0.89	0.90	0.85	0.87	1.992
	LPL	0.95	0.57	0.65	0.59	25.670
AST-based (DPY)						
DPY	LM	0.96	0.98	0.96	0.97	5
	LPL	0.96	0.53	0.54	0.53	64
Top-3 ML Models						
GBC	LM	0.862	0.385	0.148	0.214	1.072
	LPL	0.862	0.887	0.965	0.924	1.072
ADA	LM	0.848	0.349	0.232	0.279	0.682
LGB	LM	0.851	0.321	0.160	0.214	1.755
NB	LPL	0.855	0.886	0.957	0.920	0.276
QDA	LPL	0.854	0.885	0.957	0.920	0.273
Top-3 DL Models						
AutoK	LM	1.00	1.00	1.00	1.00	0.146
	LPL	1.00	1.00	1.00	1.00	0.146
MLP	LM	0.0	0.0	0.0	0.0	0.219
	LPL	0.0	0.0	0.0	0.0	0.219
CNN	LM	0.0	0.0	0.0	0.0	0.219
	LPL	0.0	0.0	0.0	0.0	0.219

5 Threats to Validity

5.0.1 External Validity. This analysis focused solely on Python, which may limit result generalizability. While Python is widely adopted, further studies are needed to verify if the observed patterns extend to other languages. Additionally, the study addressed only two code smells—LM and LPL—potentially narrowing the scope. Future work should explore other code smells to assess model performance across varying abstraction levels and broaden applicability.

5.0.2 Construct Validity. This study exclusively adopted two prompting strategies — zero-shot and CoT — to guide the models in classifying the provided code samples. Although these approaches are widely recognized in the literature, the restricted adoption of techniques may not fully reflect the inferential capabilities of SLMs. Future research could consider a broader range of prompting strategies to provide a more comprehensive understanding of model behavior in the task of code smell classification.

6 Conclusion and Next Steps

In this work, we have presented a systematic evaluation of SLMs for the detection of two pervasive Python code smells—Long Method (LM) and Long Parameter List (LPL)—and compared their performance against traditional AST-based analysis, classical ML and DL approaches. Our experiments demonstrate that SLMS showed a good performance when analyzing and detecting the code smells LM and LPL, showing they are efficient and capable of detecting the code smells analyzed.

In the detection of smells using the zero-shot as the base prompt, Mistral obtained the best performance for both LM and LPL, standing out for LM with an accuracy value of 83%, Recall 68% and F1-Score 68%. CoT without and with context prompts only manage to outperform the Zero-shot (Baseline) for Mistral with an F1-score of 87% and 85%, respectively.

When comparing the ML and DL models with SLM, we noticed that AutoK performed very well for both LM and LPL. Although ML didn't perform very well for LM, for LPL it performed better than Mistral.

As next steps, we will evaluate the interpretability of SLM and learning-generated explanations in cases where their predictions diverge from ground truth labels. When SLMs and ML/DL techniques misclassify code fragments, can the reasoning provided (e.g., via chain-of-thought prompts and SHAPE) help developers understand the source of error or ambiguity? The aim is to evaluate the clarity, relevance, and reliability of LLM and ML/DL rationales, particularly in borderline or complex cases where traditional tools (*i.e.,* DPY) may also struggle. Understanding these explanations is essential for establishing trust in SLM-assisted software quality tasks.

Data Availability

The artifacts of this study are publicly available in the Zenodo[13] and GitHub[14] repositories, to support future research.

Acknowledgements

This research was partially funded by the Brazilian funding agencies CAPES (Grant 88881.879016/2023-01), FAPESP (Grant 2023/00811-0). We also acknowledge the company Stone[15] for their support.

[13]https://zenodo.org/records/15514693
[14]https://anonymous.4open.science/r/ml4smells-core-1010
[15]https://www.stone.com.br/

References

[1] 2025. Code Contract to Send to API – Field Descriptions. https://anonymous.4open.science/r/ml4smells-core-1010/docs/contracts/code-smell-analysis-contract.md

[2] 2025. Code Extractor API Contract – Field Descriptions. https://anonymous.4open.science/r/ml4smells-core-1010/docs/contracts/code_extractor.md

[3] 2025. PySmell. https://anonymous.4open.science/r/ml4smells-core-1010

[4] Muhammad Ilyas Azeem, Fabio Palomba, Lin Shi, and Qing Wang. 2019. Machine learning techniques for code smell detection: A systematic literature review and meta-analysis. *Information and Software Technology* 108 (2019), 115–138.

[5] S. Balakrishnama and A. Ganapathiraju. 1998. Linear discriminant analysis-a brief tutorial. *Institute for Signal and information Processing* 18, 1998 (1998), 1–8.

[6] Robert C. Martin (Uncle Bob). 2017. *Solid Relevance.* Ph. D. Dissertation. https://blog.cleancoder.com/uncle-bob/2020/10/18/Solid-Relevance.html

[7] Aryan Boloori and Tushar Sharma. [n. d.]. DPy: Code Smells Detection Tool for Python. ([n. d.]).

[8] Aryan Boloori and Tushar Sharma. 2025. DPy: Code Smells Detection Tool for Python. In *Proceedings of the 22nd International Conference on Mining Software Repositories (MSR 2025) – Data and Tool Showcase Track*. Ottawa, Canada. https://2025.msrconf.org/details/msr-2025-data-and-tool-showcase-track/2/DPy-Code-Smells-Detection-Tool-for-Python

[9] Zhifei Chen, Lin Chen, Wanwangying Ma, and Baowen Xu. 2016. Detecting code smells in Python programs. In *2016 international conference on Software Analysis, Testing and Evolution (SATE)*. IEEE, 18–23.

[10] Zhifei Chen, Lin Chen, Wanwangying Ma, and Baowen Xu. 2016. Detecting Code Smells in Python Programs. In *2016 International Conference on Software Analysis, Testing and Evolution (SATE)*. 18–23. https://doi.org/10.1109/SATE.2016.10

[11] Opher Etzion and Peter Niblett. 2010. *Event Processing in Action.* Manning Publications, Shelter Island, NY.

[12] Junliang Fan, Xin Ma, Lifeng Wu, Fucang Zhang, Xiang Yu, and Wenzhi Zeng. 2019. Light Gradient Boosting Machine: An efficient soft computing model for estimating daily reference evapotranspiration with local and external meteorological data. *Agricultural water management* 225 (2019), 105758.

[13] Eduardo Fernandes, Johnatan Oliveira, Gustavo Vale, Thanis Paiva, and Eduardo Figueiredo. 2016. A review-based comparative study of bad smell detection tools. In *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering* (Limerick, Ireland) *(EASE '16)*. Association for Computing Machinery, New York, NY, USA, Article 18, 12 pages. https://doi.org/10.1145/2915970.2915984

[14] Python Software Foundation. 2024. ast — Abstract Syntax Trees. https://docs.python.org/3/library/ast.html#module-ast Accessed on: December 24, 2024.

[15] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. 1999. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, Boston, MA, USA. https://martinfowler.com/books/refactoring.html

[16] Yoav Freund and Robert E Schapire. 1997. A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of computer and system sciences* 55, 1 (1997), 119–139.

[17] Jerome H Friedman. 2001. Greedy function approximation: a gradient boosting machine. *Annals of statistics* (2001), 1189–1232.

[18] Pierre Geurts, Damien Ernst, and Louis Wehenkel. 2006. Extremely randomized trees. *Machine learning* 63 (2006), 3–42.

[19] Amid Golmohammadi, Man Zhang, and Andrea Arcuri. 2023. Testing RESTful APIs: A Survey. *ACM Trans. Softw. Eng. Methodol.* 33, 1, Article 27 (Nov. 2023), 41 pages. https://doi.org/10.1145/3617175

[20] Ruchin Gupta, Narendra Kumar, Sunil Kumar, and Jitendra Kumar Seth. 2024. Unsupervised Machine Learning for Effective Code Smell Detection: A Novel Method. *Journal of Communications Software and Systems* 20, 4 (2024), 307–316.

[21] Aurélien Géron. 2019. Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems. (2019).

[22] Arthur E Hoerl and Robert W Kennard. 1970. Ridge regression: Biased estimation for nonorthogonal problems. *Technometrics* 12, 1 (1970), 55–67.

[23] Gregor Hohpe and Bobby Woolf. 2003. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions.* Addison-Wesley Professional, Boston, MA. https://www.amazon.com/dp/0321200683

[24] Wenhua Hu, Lei Liu, Peixin Yang, Kuan Zou, Jiajun Li, Guancheng Lin, and Jianwen Xiang. 2023. Revisiting" code smell severity classification using machine learning techniques". In *47th Annual Computers, Software, and Applications Conference (COMPSAC)*. 840–849.

[25] Nasraldeen Alnor Adam Khleel and Károly Nehéz. 2024. Improving accuracy of code smells detection using machine learning with data balancing techniques. *The Journal of Supercomputing* 80, 14 (2024), 21048–21093.

[26] Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2023. Large Language Models are Zero-Shot Reasoners. arXiv:2205.11916 [cs.CL] https://arxiv.org/abs/2205.11916

[27] Hui Liu, Jiahao Jin, Zhifeng Xu, Yanzhen Zou, Yifan Bu, and Lu Zhang. 2019. Deep learning based code smell detection. *IEEE transactions on Software Engineering* 47, 9 (2019), 1811–1837.

[28] Mark Masse. 2011. *REST API design rulebook.* " O'Reilly Media, Inc.".

[29] Himesh Nanadani, Mootez Saad, and Tushar Sharma. 2023. Calibrating Deep Learning-based Code Smell Detection using Human Feedback. In *23rd International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 37–48.

[30] Ollama. 2025. Ollama. https://ollama.com/ Accessed on: January 2, 2025.

[31] Thanis Paiva, Amanda Damasceno, Eduardo Figueiredo, and Cláudio Sant'Anna. 2017. On the evaluation of code smells and detection tools. *Journal of Software Engineering Research and Development* 5, 1 (2017), 7. https://doi.org/10.1186/s40411-017-0041-1

[32] Yingli Qin. 2018. A review of quadratic discriminant analysis for high-dimensional data. *Wiley Interdisciplinary Reviews: Computational Statistics* 10, 4 (2018), e1434.

[33] Dean Wampler Robert C. Martin (Uncle Bob). 2008. *Clean code.* Pearson.

[34] Rana Sandouka and Hamoud Aljamaan. 2023. Python code smells detection using conventional machine learning models. *PeerJ Computer Science* 9 (2023), e1370. https://doi.org/10.7717/peerj-cs.1370 https://doi.org/10.7717/peerj-cs.1370.

[35] Geanderson Santos, Amanda Santana, Gustavo Vale, and Eduardo Figueiredo. 2023. Yet Another Model! A Study on Model's Similarities for Defect and Code Smells. In *Fundamental Approaches to Software Engineering*. 282–305.

[36] Amazon Web Services. 2024. Anti-Corruption Layer (ACL). https://docs.aws.amazon.com/prescriptive-guidance/latest/cloud-design-patterns/acl.html Accessed on: December 24, 2024.

[37] Luciana Lourdes Silva, Janio Rosa da Silva, Joao Eduardo Montandon, Marcus Andrade, and Marco Tulio Valente. 2024. Detecting Code Smells using ChatGPT: Initial Insights. In *Proceedings of the 18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement* (Barcelona, Spain) *(ESEM '24)*. Association for Computing Machinery, New York, NY, USA, 400–406. https://doi.org/10.1145/3674805.3690742

[38] Shreyas Subramanian, Vikram Elango, and Mecit Gungor. 2025. Small Language Models (SLMs) Can Still Pack a Punch: A survey. arXiv:2501.05465 [cs.CL] https://arxiv.org/abs/2501.05465

[39] Nikolaos Tsantalis, Theodoros Chaikalis, and Alexander Chatzigeorgiou. 2008. JDeodorant: Identification and Removal of Type-Checking Bad Smells. In *2008 12th European Conference on Software Maintenance and Reengineering*. 329–331. https://doi.org/10.1109/CSMR.2008.4493342

[40] Santiago A. Vidal, Claudia Marcos, and J. Andrés Díaz-Pace. 2016. An approach to prioritize code smells for refactoring. *Automated Software Engg.* 23, 3 (Sept. 2016), 501–532. https://doi.org/10.1007/s10515-014-0175-x

[41] Fali Wang, Zhiwei Zhang, Xianren Zhang, Zongyu Wu, Tzuhao Mo, Qiuhao Lu, Wanjing Wang, Rui Li, Junjie Xu, Xianfeng Tang, Qi He, Yao Ma, Ming Huang, and Suhang Wang. 2024. A Comprehensive Survey of Small Language Models in the Era of Large Language Models: Techniques, Enhancements, Applications, Collaboration with LLMs, and Trustworthiness. arXiv:2411.03350 [cs.CL] https://arxiv.org/abs/2411.03350

[42] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. 2023. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. arXiv:2201.11903 [cs.CL] https://arxiv.org/abs/2201.11903

[43] Stewart W Wilson. 2002. Classifiers that approximate functions. *Natural Computing* 1, 2 (2002), 211–234.

[44] Di Wu, Fangwen Mu, Lin Shi, Zhaoqiang Guo, Kui Liu, Weiguang Zhuang, Yuqi Zhong, and Li Zhang. 2024. iSMELL: Assembling LLMs with Expert Toolsets for Code Smell Detection and Refactoring. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering* (Sacramento, CA, USA) *(ASE '24)*. Association for Computing Machinery, New York, NY, USA, 1345–1357. https://doi.org/10.1145/3691620.3695508

[45] Zhifei Chen; Lin Chen; Wanwangying Ma; Baowen Xu. 2025. Pysmell. https://github.com/chenzhifei731/Pysmell. Acesso em: 28 abr. 2025.

[46] Pravin Singh Yadav, Rajwant Singh Rao, Alok Mishra, and Manjari Gupta. 2024. Machine learning-based methods for code smell detection: a survey. *Applied Sciences* 14, 14 (2024), 6149.

[47] Dongwen Zhang, Shuai Song, Yang Zhang, Haiyang Liu, and Gaojie Shen. 2023. Code Smell Detection Research Based on Pre-training and Stacking Models. *Latin America Transactions* 22, 1 (2023), 22–30.

[48] Haiyin Zhang, Luís Cruz, and Arie Van Deursen. 2022. Code smells for machine learning applications. In *Proceedings of the 1st international conference on AI engineering: software engineering for AI*. 217–228.

Received 14 May 2025; revised 08 July 2025; accepted 15 July 2025