

Article

Leveraging Code Language Models for Code Smells Detection

Soumia Zellagui ^{1,*}, Abderrezak Kenniche ¹, Samiha Fadloun ²  and Souham Meshoul ^{3,*} 

¹ Laboratoire de la Communication dans les Systèmes Informatiques (LCSI), École Nationale Supérieure d'Informatique (ESI), Oued Smar, Algiers 16309, Algeria; ka_kenniche@esi.dz

² Laboratoire de Méthodes de Conception des Systèmes (LMCS), École Nationale Supérieure d'Informatique (ESI), Oued Smar, Algiers 16309, Algeria; s_fadloun@esi.dz

³ Department of Information Technology, College of Computer and Information Sciences, Princess Nourah Bint Abdulrahman University, Riyadh 11671, Saudi Arabia

* Correspondence: s_zellagui@esi.dz (S.Z.); sbmeshoul@pnu.edu.sa (S.M.)

Abstract

Improving software quality remains a major challenge in modern software engineering. Among the different techniques used to address this issue, code smell detection plays an important role, as code smells may negatively affect software maintainability, readability, and evolution. Recent advances in machine learning, particularly pre-trained code representation models, have opened new perspectives for automating this task. In this study, code smell detection is formulated as a multi-label classification problem in order to reflect realistic scenarios where multiple smells may co-occur within the same code fragment. To enable the evaluation of pre-trained code representation models in a multi-label setting, six multi-label datasets containing source code were derived from two existing datasets, namely MLCQ and Fontana, by merging instances sharing common code smells. The study investigates multiple experimental configurations based on CodeBERT, GraphCodeBERT, and CodeT5, including direct transformer-based classification, sliding window processing for long code sequences, and hybrid approaches combining transformer embeddings with classical machine learning classifiers. Among the evaluated configurations, the hybrid approach based on CodeBERT embeddings combined with a Random Forest classifier achieved the best overall performance, reaching an accuracy of 0.91 and a micro-F1 score of 0.90 on the Fontana datasets.

Keywords: code smell; machine learning; CodeBERT; GraphCodeBERT; CodeT5

1. Introduction

Software quality has been a major concern for many years. It involves enhancing the quality of software either during the early stages of design by applying sound engineering practices and design patterns or after the code has been written, in order to improve its maintainability and future evolution. Among the techniques used to support this goal, one widely adopted approach is the detection of code smells [1].

A code smell refers to an anomaly in the source code that suggests a violation of fundamental design principles such as abstraction, encapsulation, modularity, or hierarchy [2]. The term code smells (or bad code smells) was originally introduced by Kent Beck and later popularized by Martin Fowler in his seminal work on refactoring [3]. These smells do not necessarily indicate bugs, but they often signal deeper design issues that can hinder code maintainability and evolution. Code smells can appear at different levels of the software structure, mainly at the class level and the method level. Class-level smells, such



Received: 6 June 2026

Revised: 29 June 2026

Accepted: 9 July 2026

Published: 13 July 2026

Copyright: © 2026 by the authors.

Licensee MDPI, Basel, Switzerland.

This article is an open access article distributed under the terms and conditions of the [Creative Commons Attribution \(CC BY\)](https://creativecommons.org/licenses/by/4.0/) license.

as *God Class*, reflect poor class design and excessive responsibilities, while method-level smells, such as *Long Method* or *Switch Statement*, indicate issues in the implementation of individual methods.

The impact of code smells on software quality has been extensively studied by various researchers (e.g., [4,5]), particularly through the analysis of maintenance effort. Their findings indicate that code affected by code smells tend to undergo more frequent changes and require greater maintenance effort. As software maintainability becomes an increasingly critical concern, code smell detection has attracted considerable attention. According to Kessentini et al. [6], the existing approaches can be categorized into seven groups: cooperative-based, visualization-based, machine learning-based, probabilistic, metric-based, symptoms-based, and manual approaches. Among them, metric-based approaches are the most traditional and widely integrated in modern IDEs and quality assurance tools. However, a major limitation of these approaches lies in their reliance on predefined thresholds, which are often subjective and may vary significantly across development teams. For this reason, machine learning-based approaches have emerged as a promising alternative.

Machine learning-based approaches can be classified into three categories: binary, multi-class, and multi-label classification. In the context of code smell detection, multi-label classification appears to be the most appropriate, as a single piece of code can exhibit multiple types of code smells simultaneously [7,8]. For example, in [8], in a dataset containing 40,888 instances of 13 different code smells identified across the 395 releases of 30 software systems, 24,124 of the smelly classes (59%) are affected by more than one type of code smell. Therefore, a multi-label classification approach is essential.

Recently, pre-trained code representation models such as CodeBERT, GraphCodeBERT, and CodeT5 have shown promising results in various software engineering tasks. These models generate contextual embeddings that capture both syntactic and semantic information from source code, potentially providing richer representations than traditional metrics (e.g., McCabe metrics [9] and CK metrics [10]). However, their use in multi-label code smell detection remains limited. In particular, there is a lack of systematic studies evaluating their performance in realistic multi-label settings, as well as a lack of suitable datasets that capture label co-occurrence.

In this paper, we address these limitations by formulating code smell detection as a multi-label classification problem and by preparing a multi-label dataset through the integration of existing benchmarks. This dataset enables a more realistic evaluation by preserving the co-occurrence of multiple smells within the same instances. We then conduct a systematic empirical study to evaluate the effectiveness of pre-trained code models (CodeBERT, GraphCodeBERT, and CodeT5) combined with standard classifiers. The main contributions of this work are as follows:

1. A systematic empirical evaluation of pre-trained code representation models for multi-label code smell detection under different experimental settings.
2. The preparation of multi-label datasets from existing benchmarks to enable the experimental evaluation, since, to the best of our knowledge, no publicly available multi-label code smell dataset containing source code was available for the considered smell combinations.

The remainder of this paper is organized as follows: Section 2 reviews related work. Sections 3, 3.1, and 3.2 describe our proposed methodology, which consists of two main phases: dataset preparation and model training. Section 4 details the experimental setup and discusses the obtained results and Section 5 concludes the paper and outlines potential future work.

2. Related Work

Early approaches to code smell detection relied on manually defined heuristics and thresholds derived from software metrics. While still used in practice, these methods suffer from limited generalization and strong dependency on expert knowledge. To overcome these limitations, machine learning and, more recently, deep learning approaches have been introduced, enabling models to automatically learn patterns associated with code smells. Several surveys review these approaches, focusing on models that rely on software metrics or learned representations from source code [11–18]. Within this paradigm, several studies have explored improvements to traditional machine learning techniques, particularly through ensemble learning, feature selection, and data balancing strategies. For example, Rao et al. [19] and Yadav et al. [20] show that combining multiple classifiers can enhance detection accuracy and robustness. However, these approaches are typically formulated as binary or multi-class classification problems and do not account for the co-occurrence of multiple code smells within the same code element.

In this work, we focus on studies that either formulate code smell detection as a multi-label classification problem or leverage pre-trained code representation models, as these directions better reflect real-world scenarios and recent advances in code understanding.

Multi-label code smell detection: to account for the fact that multiple code smells may co-occur within the same code element, some studies have modeled code smell detection as a multi-label classification problem. Guggulothu et al. [2] were among the first to explore this direction by applying multi-label learning strategies such as Binary Relevance, Classifier Chains, and Label Combination. Their results showed that modeling label dependencies can improve performance. Li et al. [21] further extended this idea by proposing a hybrid deep learning model combining Graph Convolutional Networks and BiLSTM to detect multiple code smells simultaneously.

Pre-trained code models and LLM-based approaches: recent advances in pre-trained code representation models and large language models (LLMs) have opened new perspectives for code smell detection. These models generate contextual embeddings that capture both syntactic and semantic properties of source code, providing richer representations than traditional metrics. Ali et al. [22] proposed a transformer-based framework for code smell detection that leverages contextual code representations to improve detection accuracy over traditional metric-based approaches. Skipina et al. [23] evaluated embeddings generated by models such as CodeT5 and CuBERT in combination with classical classifiers. Other recent studies have explored the use of LLMs for code smell detection, including prompt-based approaches such as [24], as well as hybrid frameworks like [25,26]. In addition, domain-specific studies such as [27] highlight the adaptability of these models across different programming environments.

Despite their strong performance, most of these approaches either focus on single-label classification or do not explicitly analyze the impact of dataset characteristics such as label co-occurrence.

Research gap and positioning: Although multi-label learning and pre-trained code models have been explored separately, limited attention has been given to combining these two directions in a unified framework for code smell detection. Furthermore, the availability of suitable multi-label datasets remains limited, which restricts systematic evaluation in realistic settings.

In contrast to prior studies, this work focuses on multi-label code smell detection using pre-trained code representation models, while also introducing a multi-label dataset that integrates multiple sources to better capture label co-occurrence. This combination enables a more realistic and comprehensive evaluation setting.

3. Experimental Methodology

Our methodology is structured as an end-to-end pipeline for investigating multi-label code smell detection using pre-trained code representation models, namely CodeBERT, GraphCodeBERT, and CodeT5. Rather than introducing a new learning architecture, the objective is to evaluate how these models can be leveraged, both directly and in combination with classical machine learning techniques, for this task.

The pipeline illustrated in Figure 1 is organized into three main components: (1) data preparation, (2) representation learning using pre-trained models, and (3) classification using either fine-tuned models or embedding-based approaches.

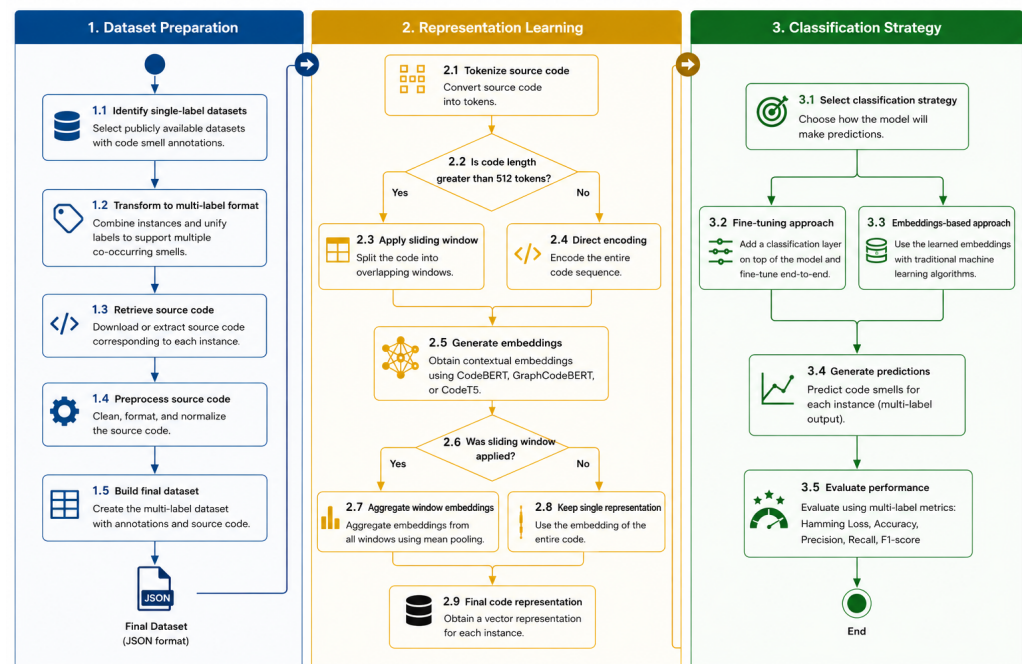


Figure 1. Experimental Framework.

The data preparation phase addresses limitations of existing metric-based datasets, particularly the absence of source code and the lack of multi-label annotations. To this end, existing datasets are transformed into multi-label datasets and enriched with the corresponding source code retrieved from GitHub (<https://github.com/>). Last accessed: 26 June 2026). A series of preprocessing steps is then applied to ensure consistency and remove non-informative elements.

For representation learning, pre-trained code models are used to encode source code into dense vector representations. Two strategies are explored. First, the models are fine-tuned directly for multi-label classification. Second, embeddings extracted from these models are used as input features for classical machine learning classifiers. The hybrid approach combines the semantic representation power of pre-trained code models with the efficiency of classical machine learning classifiers. Code language models requires substantial computational resources and large training datasets, whereas embedding extraction followed by classifiers allows faster training, lower computational cost, and improved robustness under limited hardware conditions.

The following sections describe each component of the pipeline in detail.

3.1. Dataset Preparation

A key challenge in this work is the absence of readily usable multi-label datasets for code smell detection. It is important to note that some existing studies such as [2,21,28]

rely on multi-label datasets that are not publicly available or do not provide access links, which limits reproducibility. As a result, these datasets cannot be directly reused or fairly compared against, making experimental validation difficult.

More generally, most available code smell datasets are either metric-based (without source code) or limited to single-label annotations, which further restricts their suitability for embedding-based and multi-label approaches.

To overcome these limitations, we construct our own multi-label datasets by transforming existing single-label datasets and enriching them with the corresponding source code. This choice enables both reproducibility and compatibility with pre-trained code representation models.

The overall data preparation process follows a structured pipeline, including: (i) defining the scope of considered code smells, (ii) selecting and transforming datasets into a multi-label setting, (iii) retrieving the corresponding source code, (iv) analyzing label distributions, and (v) preprocessing the data.

3.1.1. Code Smell Definition and Scope

We begin by defining the scope of code smells considered in this study, as it directly determines the granularity of analysis and labeling. In this work, we adopt Fowler's definitions [3] as a conceptual reference.

Code smells are categorized based on their scope into:

- *Class-level smells*, which affect the structure and responsibilities of classes (e.g., God Class, Data Class),
- *Method-level smells*, which concern the implementation of individual methods (e.g., Long Method, Feature Envy).

The experiments focus on the following smells:

- **Class-level:**
 - *God Class (GC)*: a class that accumulates excessive responsibilities.
 - *Data Class (DC)*: a class containing mostly data fields and accessors.
- **Method-level:**
 - *Long Method (LM)*: a method that is excessively long and complex.
 - *Feature Envy (FE)*: a method that relies more on other classes than its own.
 - *Long Parameter List (LPL)*: a method that takes too many parameters, often indicating poor encapsulation or a need to group data into objects
 - *Switch Statement (SS)*: multiple switch or conditional statements based on the same variable, leading to duplicated logic.

3.1.2. Multi-Label Dataset Construction

After defining the scope, the next step consists in constructing a suitable dataset. Two strategies can be considered:

- building datasets from scratch using rule-based tools (e.g., JSpirit [29], JDeodorant [30]) followed by manual validation,
- transforming existing single-label datasets into multi-label ones.

The first approach, although capable of producing high-quality annotations, is time-consuming, difficult to scale, and introduces subjectivity. Therefore, we adopt the second strategy, which enables the reuse of established benchmarks while ensuring scalability and reproducibility.

According to Nasrabadi et al. [31], 45 code smell datasets have been identified, among which only 25 are publicly available. However, many of these datasets could not be used due to:

- broken or inaccessible download links,
- unsupported formats,
- absence of source code references required for embedding-based approaches.

As a result, only a limited number of datasets could be successfully processed. Based on these constraints, we selected two datasets:

- The Madeyski Lewowski Code Quest (MLCQ) dataset [32].
- The dataset proposed by Fontana et al. [33], hereafter referred to as the Fontana dataset.

Both datasets are originally single-label and based on Java source code. Inspired by the multi-label formulation introduced by Guggulothu et al. [2], these datasets were transformed into multi-label datasets by merging annotations that refer to the same software entity. However, our construction process differs from their approach. In particular, we did not use an additional classifier to infer missing labels for non-common instances. Instead, when a smell annotation was not observed for a matched entity in the original datasets, the corresponding label was considered absent. The conversion process differs slightly between the two datasets. For MLCQ, instances referring to the same software entity were identified using the GitHub URL provided in the original dataset. When multiple smell annotations were associated with the same URL, the corresponding labels were merged into a single multi-label representation. For Fontana, matching relied on the metadata available in the original records, including the project identifier, project version, package name, class name, and, for method-level smells, the method signature. When multiple annotations referred to the same software entity, their labels were merged into a single multi-label instance.

3.1.3. Source Code Retrieval

Once the datasets are selected and transformed, the next step consists in retrieving the corresponding source code required for embedding-based approaches.

Pre-trained code representation models require access to raw source code. However, the selected datasets do not always provide direct access to the corresponding source code. To address this limitation, source code files are retrieved from GitHub repositories using:

- *Blob URLs*, which provide a web-based visualization,
- *Raw URLs*, which return the plain-text content of the source code.

The raw URLs are used for automated retrieval, as they allow direct access to the code without HTML formatting.

3.1.4. Multi-Label Dataset Statistics

After dataset construction and source code retrieval, we analyze the distribution of labels to better understand dataset characteristics.

Tables 1 and 2 present the distribution of instances across different combinations of code smells. As can be observed in Table 1, 70 instances are simultaneously annotated as both *Data Class* and *God Class*, although these code smells are conceptually distinct. This situation originates from the characteristics of the original datasets, where code smells were manually assessed by reviewers based on perceived code quality. According to human judgment, certain code elements may simultaneously exhibit characteristics associated with both smells. For example, certain utility or UI-related classes may contain numerous attributes and accessor methods typical of a *Data Class*, while also centralizing multiple responsibilities such as event handling, layout management, or component coordination, which are characteristic of a *God Class*. As a result, reviewer-based labeling may assign both labels to the same class (An example is the class `TextAndButtonSection` from the Eclipse BIRT project).

Table 1. Distribution of Labels Across MLCQ Dataset.

Dataset (Label ₁ , Label ₂)	Label ₁	Label ₂	Both	None
(Data Class, God Class)	271	240	70	1437
(Feature Envy, Long Method)	67	154	69	1734

Table 2. Distribution of Labels Across Fontana Dataset.

Dataset (Label ₁ , Label ₂)	Label ₁	Label ₂	Both	None
(Data Class, God Class)	132	137	0	155
(Feature Envy, Long Method)	54	53	83	245
(Long Parameters List, Switch Statement)	114	112	20	519

In addition to the three pairwise multi-label datasets derived from Fontana (*DC + GC*, *FE + LM*, and *LPL + SS*), a fourth dataset combining the four method-level smells (*FE*, *LM*, *LPL*, and *SS*) was constructed. This dataset was obtained by merging software entities appearing in the *FE*, *LM*, *LPL*, and *SS* datasets using the same matching procedure described previously. Each resulting instance was represented by a four-dimensional label vector indicating the presence or absence of each smell. Consequently, the dataset enables the evaluation of models in a higher-dimensional multi-label setting compared to the pairwise configurations. This dataset is statistically summarized in Table 3.

In Table 3, the columns *FE*, *LM*, *LPL*, and *SS* correspond to instances exhibiting only Feature Envy, Long Method, Long Parameter List, and Switch Statement, respectively. The columns (*FE*, *LM*) and (*LPL*, *SS*) denote instances simultaneously exhibiting both code smells of the corresponding pair. The *None* column represents instances that do not exhibit any of the four considered code smells.

Table 3. Distribution of Labels Across the Four-Label Fontana Dataset.

Dataset (Label ₁ , ..., Label _n)	FE	LM	LPL	SS	(LPL, SS)	(FE, LM)	None
(FE, LM, LPL, SS)	54	53	114	112	20	83	440

To support reproducibility, both the original and final datasets are made publicly available through a GitHub repository (<https://github.com/SoumiaZellagui/MLD.git>. Last accessed: 26 June 2026).

3.1.5. Data Preprocessing

Finally, before model training, a preprocessing pipeline is applied to improve data quality and remove irrelevant information.

- removal of tabs and redundant whitespace,
- removal of comments (single-line, multi-line, JavaDoc),
- removal of package declarations, imports statements and class annotations,
- removal of logging and print statements,
- removal of overridden `toString()` methods at class level.

These steps aim to reduce noise and retain only structural and semantic information relevant to code smell detection.

Each dataset is stored in JSON format as depicted in Listing 1:

Listing 1. Example of a dataset entry.

```
{
  'source_code': 'public void m(){ ... }',
  'code_smells': [
    'LongMethod', 'FeatureEnvy'
  ]
}
```

3.2. Representation Learning and Classification

Following the dataset preparation phase, the objective is to leverage pre-trained code representation models for multi-label code smell detection. Rather than introducing a new learning architecture, this study focuses on evaluating how existing models can be effectively applied under realistic constraints.

The overall learning pipeline consists of three main stages: (i) code representation using pre-trained models, (ii) classification using either fine-tuning or embedding-based approaches, and (iii) training optimization and evaluation.

3.2.1. Code Representation Using Pre-Trained Models

The prepared datasets are used to generate vector representations of source code using three pre-trained models: CodeBERT [34], GraphCodeBERT [35], and CodeT5 [36].

- *CodeBERT*: is a pre-trained model trained on programming languages (e.g., Python, Java, JavaScript, and Go) and natural language. It was trained on more than six million code snippets from GitHub repositories. The model is designed to understand the relationship between code and its corresponding natural language descriptions. CodeBERT generates contextual embeddings for individual tokens as well as a global representation for the entire code snippet, enabling it to capture both syntactic and semantic properties. These embeddings are effective for downstream tasks such as code classification, code search, and code summarization.
- *GraphCodeBERT*: extends CodeBERT by incorporating structural information in the form of data flow graphs. In addition to code token sequences, the model also takes data flow edges as input, capturing variable dependencies within the code. It was trained on over 83,000 open-source repositories and evaluated on several code-related tasks, including code search, code summarization, clone detection, and code refinement. The integration of structural information through graph representation improves the model's ability to understand the deeper behavior of code.
- *CodeT5*: is a encoder-decoder model for programming and natural languages, built upon the T5 (Text-to-Text Transfer Transformer) architecture. It is trained on a large corpus that includes both source code and natural language, covering tasks such as code summarization, code generation, translation, and defect detection. Unlike CodeBERT, which is encoder-only, CodeT5 leverages a full encoder-decoder structure, making it particularly suitable for generative tasks. It outputs both token-level and sequence-level embeddings that can be used for both classification and generation tasks. CodeT5 supports multiple programming languages including Python, Java, JavaScript, Ruby, Go, and PHP.

3.2.2. Tokenization

To convert raw code into model-compatible inputs, tokenization is applied. This process transforms source code into sequences of tokens represented as numerical identifiers (`input_ids`), along with corresponding attention masks. Special tokens are introduced to structure the input, including `[CLS]`, a classification token added at the beginning of the

sequence; [SEP], used to separate segments or mark the end of a sequence; and [PAD], a padding token used to ensure uniform sequence length within a batch [34]. The tokenizer first splits the source code into smaller units (tokens) and inserts the appropriate special tokens at their designated positions. Each token is then mapped to a unique identifier, and an *attention_mask* is generated, where real tokens are assigned a value of 1 and padding tokens a value of 0. Finally, sequences are either truncated or padded to match the model's maximum input length, typically 512 tokens [34].

Since these models impose a maximum input length of 512 tokens, long code elements are handled using a *sliding window* strategy. Code is split into overlapping segments, each processed independently. The resulting embeddings are then aggregated using pooling techniques. Common pooling techniques include *mean pooling* (average across tokens), *max pooling* (maximum values), *sum pooling* (summation), and *min pooling* (minimum values). These methods summarize the distributed representations into a compact form that retains the most relevant information. Mean pooling is often preferred for its balance between stability and informativeness [37].

The application of the sliding window strategy significantly increases computational cost, as each code instance is divided into multiple overlapping segments processed independently by the model. Consequently, the number of forward passes grows proportionally to the number of generated windows. To ensure computational feasibility, the following constraints were introduced in the processing pipeline:

- *Length Filtering*: Code instances exceeding 10,000 tokens were excluded, as extremely long sequences lead to excessive processing time and memory consumption, and may yield less reliable aggregated representations.
- *Dataset Reduction*: A fixed number of 40 instances was randomly selected for each label combination, resulting in a reduced yet balanced dataset. This allows tractable training while preserving representative label distributions.

These design choices enable the application of sliding window processing under practical constraints, while maintaining a balance between computational efficiency and data representativeness.

3.2.3. Multi-Label Classification Strategy

Given that a single code element may exhibit multiple code smells, the task is formulated as a multi-label classification problem. Two complementary strategies are explored:

- *Fine-tuning strategy*: pre-trained models are directly fine-tuned on the multi-label datasets by adding a classification layer on top of the encoder.
- *Embedding-based strategy*: embeddings extracted from pre-trained models are used as input features for classical machine learning classifiers: Random Forest, Decision Tree and SVM.

The second strategy offers a practical advantage in terms of computational efficiency. While fine-tuning large models requires significant resources, using embeddings with classical classifiers enables faster training and reduces hardware requirements, while maintaining competitive performance.

3.2.4. Label Encoding

Although labels are initially stored as lists of code smells (e.g., [“Long Method”, “Feature Envy”]), machine learning models require numerical representations.

Therefore, labels are encoded as binary vectors, where each position corresponds to a specific code smell:

- 1 indicates the presence of the smell,

- 0 indicates its absence.

An example of encoded labels is shown in Listing 2:

Listing 2. Example of a dataset entry with encoded labels (See Listing 1).

```
{
  "source_code": "public void m(){ ... }",
  "code_smells": [
    "1", "1"
  ]
}
```

3.2.5. Training Strategy

Each dataset is split into training and testing sets using an 80/20 ratio, ensuring sufficient data for learning while preserving a representative evaluation set. Additionally, stratified sampling is applied during the split. This means that each label is proportionally represented in the training and testing sets (e.g., approximately 80% of instances of each label are used for training and 20% for testing), thereby reducing sampling bias and preserving the original label distribution across subsets.

3.2.6. Loss Function

To handle multi-label classification, Binary Cross-Entropy (BCE) loss is used. This loss function treats each label as an independent binary classification task. It is defined as:

$$\mathcal{L}_{\text{BCE}} = -\frac{1}{C} \sum_{j=1}^C [y_j \log(p_j) + (1 - y_j) \log(1 - p_j)] \quad (1)$$

where:

- C is the number of labels,
- y_j is the ground-truth label,
- p_j is the predicted probability.

4. Results

This section presents the experimental evaluation of the multi-label code smell detection pipeline introduced in this work. The objective is to assess the effectiveness of pre-trained code representation models under different configurations and to analyze their behavior under realistic constraints. The evaluation is structured into three main experimental scenarios:

1. Baseline evaluation with CodeBERT, GraphCodeBERT & CodeT5,
2. Sliding window with mean pooling,
3. Hybrid approach using classical machine learning algorithms.

4.1. Experimental Setup

Due to limited access to dedicated hardware, publicly available platforms were leveraged for development and experimentation. Most of the model training was conducted on a GPU-enabled Google Colab environment (<https://colab.research.google.com/>, accessed on 26 June 2026), with occasional use of Kaggle notebooks (<https://www.kaggle.com/>, accessed on 26 June 2026). Local validation was performed on a MacBook with Apple Silicon. The technical details of these environments are presented in Table 4.

Table 4. Technical Specifications of Computing Environments.

Machine	CPU	RAM	GPU/Accelerator	Storage	Operating System
Google Colab	Intel Xeon	12.7 GB	NVIDIA Tesla T4 (15 GB VRAM)	112.6 GB	Linux (Debian-based)
Kaggle	Multi-core CPU	30 GB	None	77.1 GB	Linux (Ubuntu-based)
Apple MacBook M1 ¹	Apple M1 (8-core)	8 GB (Unified)	Integrated 8-core GPU	256 GB SSD	macOS 14.5

¹ Apple Inc., Cupertino, CA, USA.

4.2. Evaluation Metrics

The evaluation of multi-label classification models requires metrics that go beyond traditional accuracy, as each instance may be associated with multiple labels simultaneously. In this context, predictions are assessed based on four main components: True Positives (TP), True Negatives (TN), False Positives (FP), and False Negatives (FN). In particular, the following metrics are used:

- Hamming Loss (HL):

$$HL = \frac{1}{N \cdot L} \sum_{i=1}^N \sum_{j=1}^L \mathbb{I}\{y_{ij} \neq \hat{y}_{ij}\}$$

- Subset Accuracy (Exact Match Ratio): in the following results tables, the term *Accuracy* refers to this metric.

$$\text{Subset Accuracy} = \frac{1}{N} \sum_{i=1}^N \mathbb{I}[Y_i = \hat{Y}_i]$$

- Micro-Precision:

$$\text{Precision}_{micro} = \frac{\sum_{i=1}^L TP_i}{\sum_{i=1}^L (TP_i + FP_i)}$$

- Micro-Recall:

$$\text{Recall}_{micro} = \frac{\sum_{i=1}^L TP_i}{\sum_{i=1}^L (TP_i + FN_i)}$$

- Micro F1-Score:

$$F1_{micro} = 2 \cdot \frac{\text{Precision}_{micro} \cdot \text{Recall}_{micro}}{\text{Precision}_{micro} + \text{Recall}_{micro}}$$

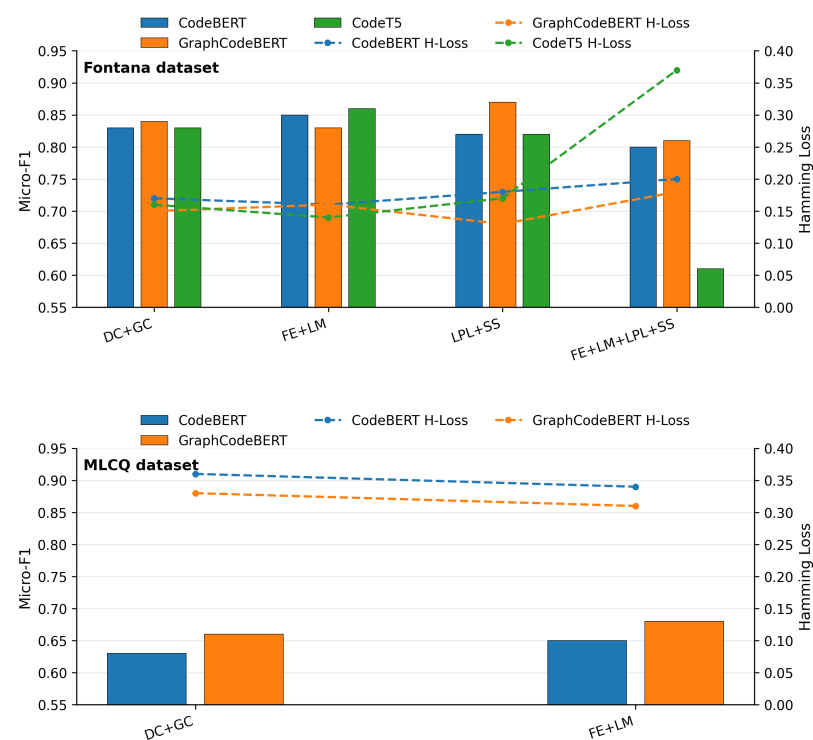
Micro-averaging is particularly suitable in our context, as it accounts for label imbalance by weighting each prediction equally across all labels.

4.3. Baseline Evaluation with CodeBERT, GraphCodeBERT, and CodeT5

In this experiment, CodeBERT, GraphCodeBERT, and CodeT5 are evaluated under the standard 512-token constraint. However, it is important to note that CodeT5 was not evaluated on the MLCQ dataset in the baseline configuration. This is due to the high computational and memory requirements of CodeT5, which made it impractical to process the longer code instances present in MLCQ under the available hardware constraints. Table 5 presents the performance of these used models and Figure 2 illustrates the corresponding micro F1-scores and Hamming Loss across all configurations and datasets. Several observations can be drawn from the results.

Table 5. Evaluation Results of Models (Baseline).

Model	Dataset	Code Smells	H-Loss	Accuracy	F1-Score	Precision	Recall
CodeBERT	Fontana	DC + GC	0.17	0.86	0.83	0.85	0.82
CodeBERT	Fontana	FE + LM	0.16	0.88	0.85	0.87	0.84
CodeBERT	Fontana	LPL + SS	0.18	0.85	0.82	0.84	0.81
CodeBERT	Fontana	FE + LM + LPL + SS	0.20	0.83	0.80	0.82	0.79
CodeBERT	MLCQ	DC + GC	0.36	0.45	0.63	0.60	0.66
CodeBERT	MLCQ	FE + LM	0.34	0.47	0.65	0.62	0.68
GraphCodeBERT	Fontana	DC + GC	0.15	0.87	0.84	0.86	0.83
GraphCodeBERT	Fontana	FE + LM	0.16	0.86	0.83	0.85	0.82
GraphCodeBERT	Fontana	LPL + SS	0.13	0.89	0.87	0.89	0.86
GraphCodeBERT	Fontana	FE + LM + LPL + SS	0.18	0.84	0.81	0.83	0.80
GraphCodeBERT	MLCQ	DC + GC	0.33	0.48	0.66	0.63	0.69
GraphCodeBERT	MLCQ	FE + LM	0.31	0.50	0.68	0.65	0.71
CodeT5	Fontana	DC + GC	0.16	0.86	0.83	0.85	0.82
CodeT5	Fontana	FE + LM	0.14	0.88	0.86	0.88	0.85
CodeT5	Fontana	LPL + SS	0.17	0.85	0.82	0.84	0.81
CodeT5	Fontana	FE + LM + LPL + SS	0.37	0.63	0.61	0.62	0.60

**Figure 2.** F1-Score and Hamming Loss of the Baseline Models.

First, all models achieve significantly higher performance on the Fontana datasets compared to MLCQ. For example, GraphCodeBERT reaches an accuracy of 0.86 and F1-score of 0.83 on (*FE + LM*), while performance drops to 0.50 accuracy and 0.68 F1-score on MLCQ (*FE + LM*). This difference is primarily due to input length constraints: Fontana instances are shorter and fully captured within the 512-token limit, whereas MLCQ instances are often truncated, leading to information loss.

Second, GraphCodeBERT generally provides competitive performance and achieves the best results in three of the four evaluated configurations as confirmed by the Fontana panel of Figure 2. This behavior can be attributed to its ability to incorporate structural information through data-flow graphs, which enhances code understanding.

Finally, performance decreases as the number of labels increases. In the ($FE + LM + LPL + SS$) setting, accuracy drops significantly (e.g., CodeT5: 0.63). In addition, in the ($FE + LM + LPL + SS$) setting, both F1 and Hamming Loss show reduced performance across all models in Figure 2, with CodeT5 showing the sharpest collapse ($F1 = 0.61$, $HL = 0.37$). This reflects the strict nature of exact match evaluation in multi-label classification.

Key Findings: Baseline

GraphCodeBERT achieves the best overall performance due to its structural awareness. Results are significantly higher on shorter datasets (Fontana), while performance drops on longer code (MLCQ) due to truncation. Increasing the number of labels further reduces accuracy because of the strict exact match criterion.

4.4. Sliding Window Evaluation with Mean Pooling

To overcome the input length limitation of transformer models, a sliding window strategy with a stride of 256 tokens was applied, allowing long code sequences to be processed in overlapping segments.

It is important to note that the MLCQ dataset was not included in this configuration. The sliding window approach significantly increases the number of forward passes required per instance, and MLCQ contains substantially longer and more complex code fragments compared to Fontana. Applying the same strategy on MLCQ would have resulted in prohibitively high computational cost and training time.

Table 6 and Figure 3 presents the performance of CodeBERT and GraphCodeBERT using the sliding window approach.

Table 6. Evaluation Results of Models with Sliding Windows and Mean Pooling.

Model	Dataset	Code Smells	H-Loss	Accuracy	F1-Score	Precision	Recall
CodeBERT	Fontana	DC + GC	0.23	0.77	0.75	0.76	0.74
CodeBERT	Fontana	FE + LM	0.21	0.79	0.77	0.78	0.76
CodeBERT	Fontana	LPL + SS	0.15	0.85	0.83	0.84	0.82
CodeBERT	Fontana	FE + LM + LPL + SS	0.18	0.81	0.79	0.80	0.78
GraphCodeBERT	Fontana	DC + GC	0.22	0.78	0.76	0.77	0.75
GraphCodeBERT	Fontana	FE + LM	0.19	0.80	0.78	0.79	0.77
GraphCodeBERT	Fontana	LPL + SS	0.13	0.86	0.84	0.85	0.83
GraphCodeBERT	Fontana	FE + LM + LPL + SS	0.17	0.82	0.80	0.81	0.79

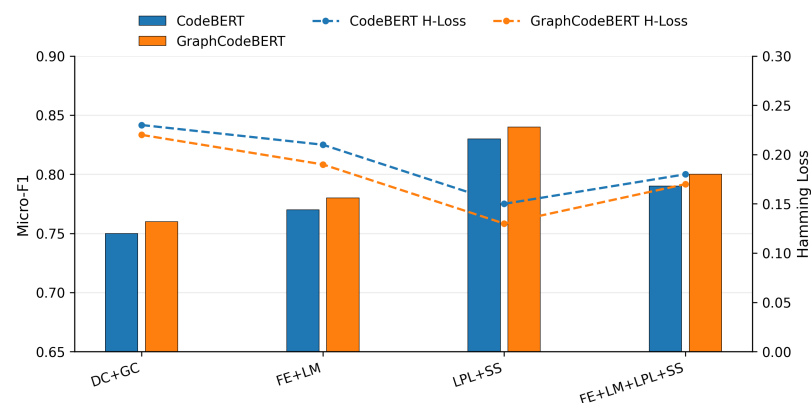


Figure 3. F1-Score and Hamming Loss of the Sliding Window Technique.

The sliding window strategy allows the models to process longer code sequences that were previously truncated in the baseline setting. Across the evaluated label combinations, the performance remains relatively consistent. The best results are obtained for the (*LPL* + *SS*) configuration, where GraphCodeBERT achieves an F1-score of 0.84, an accuracy of 0.86 and a H-Loss of 0.13. Overall, GraphCodeBERT slightly outperforms CodeBERT across all configurations.

Overall, the sliding window configuration does not lead to substantial performance gains compared with the baseline setting. In several configurations, the results remain comparable or slightly lower. For example, in the (*DC* + *GC*) configuration, performance remains moderate, reaching around 0.78 accuracy, but it still decreases compared with the corresponding baseline results. The relatively high Hamming Loss observed for this configuration (0.23 for CodeBERT and 0.22 for GraphCodeBERT), as shown in Figure 3, suggests that mean pooling may not fully preserve the local structural information required to detect these smells.

Several factors explain this behavior. First, mean pooling is used to aggregate embeddings from multiple windows. While efficient, this operation averages all representations and may dilute important local patterns. Since code smells are often associated with specific structural regions in the code, this averaging can reduce the model's ability to capture discriminative features.

Second, the sliding window strategy significantly increases computational cost. To keep the experiments feasible, the dataset was reduced through sampling (see Section 3.2). This reduction limits the diversity of training data and negatively affects the model's generalization capability.

Third, the high computational cost also restricted hyperparameter tuning. As a result, the models were trained with limited optimization, which may lead to suboptimal performance.

Key Findings: Sliding Window Evaluation

The sliding window strategy allows transformer-based models to handle longer code sequences beyond the 512-token limit. However, under limited computational resources, performance gains remain modest. GraphCodeBERT shows a slight advantage due to its ability to capture structural information, but overall performance is constrained by the limitations of mean pooling, reduced dataset size, and limited hyperparameter tuning. These results indicate that improving long-code modeling requires both better aggregation methods and sufficient computational resources.

4.5. Hybrid Approach with Classical Machine Learning

In this configuration, a hybrid approach is adopted by combining pre-trained code language model embeddings with classical machine learning classifiers. Specifically, embeddings generated by CodeBERT are used as input features for Decision Tree, Random Forest, and SVM models. For the Random Forest classifier, 150 trees were used with `max_features=log2`. For the SVM and Decision Tree classifiers, the default `scikit-learn` (version 1.2.2) parameters were used.

Although GraphCodeBERT achieved competitive results in several baseline configurations, the hybrid experiments used CodeBERT embeddings because they require a simpler extraction pipeline. Unlike CodeBERT, GraphCodeBERT incorporates data-flow information and graph-guided attention, which involves additional structural preprocessing [35]. Since runtime and memory consumption were not systematically measured, this choice should be viewed as a practical design decision.

To handle long code sequences, the sliding window strategy described in Section 4.4 is applied. Each code instance is divided into overlapping segments, encoded independently, and then aggregated using mean pooling to obtain a fixed-size representation.

Since classical machine learning algorithms are inherently designed for single-label classification, adaptation techniques are required for multi-label classification. In this work, Decision Tree and Random Forest are extended using Ensemble of Classifier Chains (ECC), while SVM is adapted using Classifier Chains (CC), allowing the models to capture label dependencies. Table 7 presents the results obtained using this hybrid approach and Figure 4 illustrates the corresponding F1-scores and Hamming Loss across all classifiers and configurations.

Table 7. Evaluation Results of ML Algorithms with Sliding Windows and Mean Pooling (Fontana Dataset).

Model	Dataset	Code Smells	H-Loss	Accuracy	F1-Score	Precision	Recall
Decision Tree	Fontana	DC + GC	0.13	0.85	0.83	0.84	0.82
Decision Tree	Fontana	FE + LM	0.12	0.87	0.85	0.86	0.84
Decision Tree	Fontana	LPL + SS	0.10	0.89	0.87	0.88	0.86
Decision Tree	Fontana	FE + LM + LPL + SS	0.14	0.74	0.84	0.82	0.86
Random Forest	Fontana	DC + GC	0.09	0.89	0.87	0.88	0.87
Random Forest	Fontana	FE + LM	0.08	0.91	0.89	0.90	0.88
Random Forest	Fontana	LPL + SS	0.06	0.91	0.90	0.91	0.89
Random Forest	Fontana	FE + LM + LPL + SS	0.11	0.86	0.85	0.86	0.84
sSVM	Fontana	DC + GC	0.11	0.87	0.85	0.86	0.85
sSVM	Fontana	FE + LM	0.10	0.89	0.87	0.88	0.86
sSVM	Fontana	LPL + SS	0.08	0.90	0.89	0.90	0.88
sSVM	Fontana	FE + LM + LPL + SS	0.13	0.84	0.83	0.84	0.82

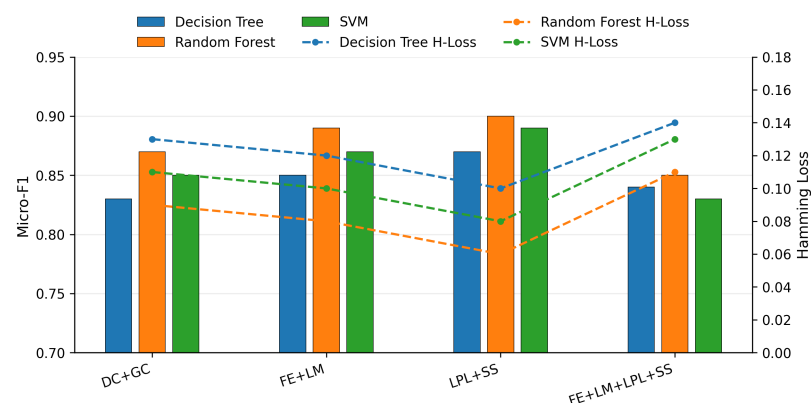


Figure 4. F1-Score and Hamming Loss of the Hybrid Approach.

Table 7 shows that Random Forest achieves the highest performance among the evaluated classifiers, followed by SVM and Decision Tree. This ranking is directly confirmed by both the bar heights and dashed H-Loss lines in Figure 4, which remain ordered across every configuration: Random Forest lowest H-Loss, Decision Tree highest. The best results are obtained for the (LPL + SS) configuration, where Random Forest reaches an accuracy of 0.91, an F1-score of 0.90 and a Hamming Loss of 0.06.

Compared with the transformer-based configurations presented in Sections 4.3 and 4.4, the hybrid approach yields higher performance under the considered experimental conditions. However, since the different configurations involve distinct preprocessing strategies and reduced datasets, these results should not be interpreted as a strictly controlled com-

parison between approaches. These observations suggest that combining CodeBERT embeddings with classical machine learning classifiers represents a promising alternative for multi-label code smell detection. In particular, Random Forest appears to exploit the generated embeddings effectively, leading to stable and competitive results.

Key Findings: Hybrid Approach with Classical ML

Among the evaluated classifiers, Random Forest combined with CodeBERT embeddings achieved the highest performance under the considered experimental conditions, reaching up to 0.91 accuracy and 0.90 micro-F1 score. These results indicate that embedding-based hybrid approaches constitute a promising alternative for multi-label code smell detection under limited computational resources.

4.6. Threats to Validity

Several threats to validity should be considered when interpreting the results of this study.

A first threat concerns the limited number of datasets used in the experiments. Only the MLCQ and Fontana datasets were considered, which may limit the generalizability of the obtained results. To partially mitigate this issue, experiments were conducted on datasets with different characteristics and label combinations. Nevertheless, future work should include additional datasets and projects from different domains to improve external validity.

Another limitation is related to dataset size. Due to computational constraints, some experiments relied on reduced datasets obtained through sampling. Although this strategy allowed the experiments to remain feasible while preserving label balance, it may reduce the diversity of training instances and affect model generalization. Furthermore, the experiments were executed once and repeated runs with different random seeds were not performed. Consequently, the reported results do not include variance estimates or statistical significance analyses. Future work could benefit from larger datasets, repeated experimental settings, and more powerful computational resources.

Another threat arises from the differences between the experimental configurations. While the baseline evaluation included both Fontana and MLCQ, the sliding-window and hybrid experiments were conducted exclusively on the Fontana dataset due to computational constraints. In addition, some experiments required sliding-window processing and dataset reduction, whereas others were conducted on the original datasets. These differences limit strict quantitative comparisons across configurations. Therefore, the reported results are interpreted cautiously and mainly used to highlight trends observed under each experimental setting. Consequently, the superior performance observed for the hybrid approach cannot be generalized to MLCQ or to other datasets without additional empirical validation.

Another threat to validity concerns the presence of potentially noisy or ambiguous annotations inherited from the original datasets. Since code smells were manually assessed by reviewers, some labels may reflect subjective interpretation. Such cases were preserved to maintain consistency with the original benchmarks and avoid introducing additional bias through manual relabeling. Additional noise may also be introduced during the multi-label dataset construction process. In particular, when an entity is present in one single-label dataset but does not appear with another smell annotation, the non-observed smell is treated as absent. However, the absence of an annotation does not necessarily guarantee the absence of the corresponding smell. Future work could investigate automated validation or consensus-based annotation strategies to reduce label ambiguity.

Another limitation concerns the programming language diversity of the datasets. All experiments were conducted exclusively on Java source code. Consequently, the obtained results may not generalize to other programming languages with different syntactic and structural properties. Future work should evaluate the proposed approach on multilingual datasets.

The sliding window strategy also introduces potential limitations. Although it enables the processing of long code sequences, the aggregation of multiple windows through mean pooling may dilute important local information associated with code smells. Alternative aggregation strategies or long-context transformer architectures could be explored in future work.

Finally, another limitation concerns the embedding model used in the hybrid configuration. Although GraphCodeBERT achieved competitive performance in the baseline experiments, only CodeBERT embeddings were combined with classical machine learning classifiers due to computational constraints. Consequently, the impact of alternative embedding models within the hybrid framework remains an open question and should be investigated in future work.

5. Conclusions

This study investigated the use of pre-trained code representation models, namely CodeBERT, GraphCodeBERT, and CodeT5, for multi-label code smell detection. The work addressed several practical challenges associated with this task, including the co-occurrence of multiple code smells, long source code sequences, and the limited availability of publicly accessible multi-label datasets containing source code. To support the experimental evaluation, existing datasets were transformed into multi-label datasets and enriched with the corresponding source code. Multiple experimental configurations were then evaluated, including direct transformer-based classification, sliding window processing for long code sequences, and hybrid approaches combining transformer embeddings with classical machine learning classifiers.

The obtained results show that pre-trained code representations are effective for multi-label code smell detection. GraphCodeBERT achieved competitive results in several baseline configurations, while the hybrid approach based on CodeBERT embeddings and Random Forest obtained the highest performance on the reduced Fontana datasets under the considered experimental conditions. These results suggest that combining transformer-based code embeddings with classical classifiers can be a promising direction, especially when computational resources are limited. Future work may explore additional programming languages, larger and more diverse datasets, alternative long-context modeling strategies, and comparisons with recent metric-based and large language model approaches. Integrating automated refactoring mechanisms could also further improve the practical applicability of the proposed approach.

Author Contributions: The S.Z. conceived the main idea, designed the methodology, and was primarily responsible for writing the paper. The A.K. constructed the dataset, implemented the proposed approach and conducted the experiments. The S.F. contributed to defining the methodology, provided supervision, and assisted in writing and proofreading the manuscript. Finally, the S.M. contributed to supervision, writing, and proofreading the manuscript. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by the Princess Nourah bint Abdulrahman University Researchers Supporting Project (No. PNURSP2026R196), Princess Nourah bint Abdulrahman University, Riyadh, Saudi Arabia.

Data Availability Statement: The datasets presented in this study are available at <https://github.com/SoumiaZellagui/MLD.git>, (accessed on 5 June 2026).

Acknowledgments: The authors would like to acknowledge the Princess Nourah bint Abdulrahman University Researchers Supporting Project number (PNURSP2026R196), Princess Nourah bint Abdulrahman University, Riyadh, Saudi Arabia.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Soh, Z.; Yamashita, A.; Khomh, F.; Guéhéneuc, Y. Do Code Smells Impact the Effort of Different Maintenance Programming Activities? In *Proceedings of the IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2016, Suita, Osaka, Japan, 14–18 March 2016*; IEEE Computer Society: Washington, DC, USA, 2016; Volume 1, pp. 393–402. [\[CrossRef\]](#)
2. Guggulothu, T.; Moiz, S.A. Code smell detection using multi-label classification approach. *Softw. Qual. J.* **2020**, *28*, 1063–1086. [\[CrossRef\]](#)
3. Fowler, M. *Refactoring: Improving the Design of Existing Code*; Addison-Wesley: Boston, MA, USA, 1999.
4. Olbrich, S.M.; Cruzes, D.S.; Basili, V.R.; Zazworka, N. The evolution and impact of code smells: A case study of two open source systems. In *Proceedings of the Third International Symposium on Empirical Software Engineering and Measurement, ESEM 2009, Lake Buena Vista, FL, USA, 15–16 October 2009*; IEEE Computer Society: Washington, DC, USA, 2009; pp. 390–400. [\[CrossRef\]](#)
5. Olbrich, S.M.; Cruzes, D.S.; Sjøberg, D.I.K. Are all code smells harmful? A study of God Classes and Brain Classes in the evolution of three open source systems. In *Proceedings of the 26th IEEE International Conference on Software Maintenance (ICSM 2010), Timisoara, Romania, 12–18 September 2010*; Marinescu, R., Lanza, M., Marcus, A., Eds.; IEEE Computer Society: Washington, DC, USA, 2010; pp. 1–10. [\[CrossRef\]](#)
6. Kessentini, W.; Kessentini, M.; Sahraoui, H.; Bechikh, S.; Ouni, A. A cooperative parallel search-based software engineering approach for code-smells detection. *IEEE Trans. Softw. Eng.* **2014**, *40*, 841–861. [\[CrossRef\]](#)
7. Hamdi, O.; Ouni, A.; AlOmar, E.A.; Mkaouer, M.W. An empirical study on code smells co-occurrences in android applications. In *Proceedings of the 2021 36th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW), Virtual, 15–19 November 2021*; IEEE: Piscataway, NJ, USA, 2021; pp. 26–33.
8. Palomba, F.; Bavota, G.; Di Penta, M.; Fasano, F.; Oliveto, R.; De Lucia, A. A large-scale empirical study on the lifecycle of code smell co-occurrences. *Inf. Softw. Technol.* **2018**, *99*, 1–10. [\[CrossRef\]](#)
9. McCabe, T.J. A complexity measure. *IEEE Trans. Softw. Eng.* **1976**, *SE-2*, 308–320. [\[CrossRef\]](#)
10. Chidamber, S.R.; Kemerer, C.F. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.* **1994**, *20*, 476–493. [\[CrossRef\]](#)
11. Azeem, M.I.; Palomba, F.; Shi, L.; Wang, Q. Machine learning techniques for code smell detection: A systematic literature review and meta-analysis. *Inf. Softw. Technol.* **2019**, *108*, 115–138. [\[CrossRef\]](#)
12. Al-Shaaby, A.; Aljamaan, H.; Alshayeb, M. Bad smell detection using machine learning techniques: A systematic literature review. *Arab. J. Sci. Eng.* **2020**, *45*, 2341–2369. [\[CrossRef\]](#)
13. Kaur, A.; Jain, S.; Goel, S.; Dhiman, G. A review on machine-learning based code smell detection techniques in object-oriented software system(s). *Recent Adv. Electr. Electron. Eng. Former. Recent Pat. Electr. Electron. Eng.* **2021**, *14*, 290–303. [\[CrossRef\]](#)
14. Grujić, K.G.; Prokić, S.; Kovačević, A.; Luburić, N.; Vidaković, D.; Slivka, J. Machine learning approaches for code smell detection: A systematic literature review. *SSRN* **2022**, 4299859. [\[CrossRef\]](#)
15. Lewowski, T.; Madeyski, L. How far are we from reproducible research on code smell detection? A systematic literature review. *Inf. Softw. Technol.* **2022**, *144*, 106783. [\[CrossRef\]](#)
16. Verma, R.; Kumar, K.; Verma, H.K. Code smell prioritization in object-oriented software systems: A systematic literature review. *J. Softw. Evol. Process* **2023**, *35*, e2536. [\[CrossRef\]](#)
17. Alazba, A.; Aljamaan, H.; Alshayeb, M. Deep learning approaches for bad smell detection: A systematic literature review. *Empir. Softw. Eng.* **2023**, *28*, 77. [\[CrossRef\]](#)
18. Yadav, P.S.; Rao, R.S.; Mishra, A.; Gupta, M. Machine learning-based methods for code smell detection: A survey. *Appl. Sci.* **2024**, *14*, 6149. [\[CrossRef\]](#)
19. Rao, R.S.; Dewangan, S.; Mishra, A. An Empirical Evaluation of Ensemble Models for Python Code Smell Detection. *Appl. Sci.* **2025**, *15*, 7472. [\[CrossRef\]](#)
20. Yadav, P.S.; Rao, R.S.; Mishra, A.; Gupta, M. Ensemble methods with feature selection and data balancing for improved code smells classification performance. *Eng. Appl. Artif. Intell.* **2025**, *139*, 109527. [\[CrossRef\]](#)
21. Li, Y.; Zhang, X. Multi-Label Code Smell Detection with Hybrid Model based on Deep Learning. In *Proceedings of the 34th International Conference on Software Engineering and Knowledge Engineering, SEKE 2022, KSIR Virtual Conference Center, Pittsburgh, PA, USA, 1 July–10 July 2022*; Peng, R., Pantoja, C.E., Kamthan, P., Eds.; KSI Research Inc.: Pittsburgh, PA, USA, 2022; pp. 42–47. [\[CrossRef\]](#)

22. Ali, I.; Rizvi, S.S.H.; Adil, S.H. Enhancing software quality with AI: A transformer-based approach for code smell detection. *Appl. Sci.* **2025**, *15*, 4559. [\[CrossRef\]](#)
23. Skipina, M.; Slivka, J.; Luburic, N.; Kovacevic, A. Automatic detection of Feature Envy and Data Class code smells using machine learning. *Expert Syst. Appl.* **2024**, *243*, 122855. [\[CrossRef\]](#)
24. Mesbah, D.; El Madhoun, N.; Al Agha, K.; Chalouati, H. Leveraging prompt-based large language models for code smell detection: A comparative study on the MLCQ dataset. In *Proceedings of the International Conference on Emerging Internet, Data & Web Technologies, Matsue, Japan, 26–28 February 2025*; Springer: Berlin/Heidelberg, Germany, 2025; pp. 444–454.
25. Ho, A.; Bui, A.M.T.; Nguyen, P.T.; Salle, A.D.; Le, B. EnseSmells: Deep ensemble and programming language models for automated code smells detection. *J. Syst. Softw.* **2025**, *224*, 112375. [\[CrossRef\]](#)
26. Wu, D.; Mu, F.; Shi, L.; Guo, Z.; Liu, K.; Zhuang, W.; Zhong, Y.; Zhang, L. 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, ASE 2024, Sacramento, CA, USA, 27 October–1 November 2024*; Filkov, V., Ray, B., Zhou, M., Eds.; ACM: New York, NY, USA, 2024; pp. 1345–1357. [\[CrossRef\]](#)
27. Vo, Q.; Dao, H.; Fukuda, K. Harnessing the Power of LLMs for Code Smell Detection in Terraform Infrastructure as Code. In *Proceedings of the 49th IEEE Annual Computers, Software, and Applications Conference, COMPSAC 2025, Toronto, ON, Canada, 8–11 July 2025*; Shahriar, H., Alam, K.S., Ohsaki, H., Cimato, S., Capretz, M.A.M., Ahmed, S., Ahamed, S.I., Majumder, A.J.A., Haque, M., Yoshihisa, T., et al., Eds.; IEEE: Piscataway, NJ, USA, 2025; pp. 533–542. [\[CrossRef\]](#)
28. Alhadeaf, H.M.; Alrashoud, M. A Unified Multi-Label Code Smell Dataset for Code Smell Detection at Different Granularities. *IEEE Access* **2026**, *14*, 714–737. [\[CrossRef\]](#)
29. Vidal, S.A.; Vázquez, H.C.; Pace, J.A.D.; Marcos, C.A.; Garcia, A.F.; Oizumi, W.N. JSPiRIT: A flexible tool for the analysis of code smells. In *Proceedings of the 34th International Conference of the Chilean Computer Science Society, SCCC 2015, Santiago, Chile, 9–13 November 2015*; IEEE: Piscataway, NJ, USA, 2015; pp. 1–6. [\[CrossRef\]](#)
30. Tsantalís, N.; Chaikalís, T.; Chatzigeorgiou, A. Ten years of JDeodorant: Lessons learned from the hunt for smells. In *Proceedings of the 25th International Conference on Software Analysis, Evolution and Reengineering, SANER 2018, Campobasso, Italy, 20–23 March 2018*; Oliveto, R., Penta, M.D., Shepherd, D.C., Eds.; IEEE Computer Society: Washington, DC, USA, 2018; pp. 4–14. [\[CrossRef\]](#)
31. Nasrabadi, M.Z.; Parsa, S.; Esmaili, E.; Palomba, F. A Systematic Literature Review on the Code Smells Datasets and Validation Mechanisms. *ACM Comput. Surv.* **2023**, *55*, 298:1–298:48. [\[CrossRef\]](#)
32. Madeyski, L.; Lewowski, T. MLCQ: Industry-Relevant Code Smell Data Set. In *Proceedings of the EASE '20: Evaluation and Assessment in Software Engineering, Trondheim, Norway, 15–17 April 2020*; Li, J., Jaccheri, L., Dingsøyr, T., Chitchyan, R., Eds.; ACM: New York, NY, USA, 2020; pp. 342–347. [\[CrossRef\]](#)
33. Fontana, F.A.; Mäntylä, M.V.; Zanon, M.; Marino, A. Comparing and experimenting machine learning techniques for code smell detection. *Empir. Softw. Eng.* **2016**, *21*, 1143–1191. [\[CrossRef\]](#)
34. Feng, Z.; Guo, D.; Tang, D.; Duan, N.; Feng, X.; Gong, M.; Shou, L.; Qin, B.; Liu, T.; Jiang, D.; et al. Codebert: A pre-trained model for programming and natural languages. *arXiv* **2020**, arXiv:2002.08155.
35. Guo, D.; Ren, S.; Lu, S.; Feng, Z.; Tang, D.; Liu, S.; Zhou, L.; Duan, N.; Svyatkovskiy, A.; Fu, S.; et al. Graphcodebert: Pre-training code representations with data flow. *arXiv* **2020**, arXiv:2009.08366.
36. Wang, Y.; Wang, W.; Joty, S.; Hoi, S.C. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv* **2021**, arXiv:2109.00859.
37. Reimers, N.; Gurevych, I. Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. In *Proceedings of the EMNLP, Hong Kong, China, 3–7 November 2019*.

Disclaimer/Publisher’s Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.