Evaluating Embedding Representations for Multiclass Code Smell Detection: A Comparative Study of CodeBERT and General-Purpose Embeddings

Marcela Mosquera [†] and Rodolfo Bojorque [*,†]

Posted Date: 20 March 2026

doi: 10.20944/preprints202603.1606.v1

Keywords: code smells; representation learning; source code embeddings; CodeBERT; multiclass classification; software quality; software maintenance

 doi:10.20944/preprints202603.1606.v1

Article

Evaluating Embedding Representations for Multiclass Code Smell Detection: A Comparative Study of CodeBERT and General-Purpose Embeddings

Marcela Mosquera [1,†] **and Rodolfo Bojorque** [2,*,†]

[1] Departamento de Informática y Ciencias de la Computación, Escuela Politécnica Nacional, Quito 170525, Ecuador
[2] Campus El Vecino, Universidad Politécnica Salesiana, Cuenca 010102, Ecuador
* Correspondence: rbojorque@ups.edu.ec
† These authors contributed equally to this work.

Abstract

Code smells are indicators of potential design problems in software systems and are commonly used to guide refactoring activities. Recent advances in representation learning have enabled the use of embedding-based models for analyzing source code, offering an alternative to traditional approaches based on manually engineered metrics. However, the effectiveness of different embedding representations for multiclass code smell detection remains insufficiently explored. This study presents an empirical comparison of embedding models for the automatic detection of three widely studied code smells: Long Method, God Class, and Feature Envy. Using the Crowdsmelling dataset as an empirical basis, source code fragments were extracted from the original projects and transformed into vector representations using two embedding approaches: a general-purpose embedding model and the code-specialized CodeBERT model. The resulting representations were evaluated using several machine learning classifiers under a stratified group-based validation protocol. The results show that CodeBERT consistently outperforms the general-purpose embeddings across multiple evaluation metrics, including balanced accuracy, macro F1-score, and Matthews correlation coefficient. Dimensionality reduction analyses using PCA and t-SNE further indicate that CodeBERT organizes code smell instances in a more structured latent representation space, which facilitates the separation of smell categories. These findings provide empirical evidence that domain-specific pretraining plays an important role in representation learning for software engineering tasks and that embedding choice significantly influences the separability of code smell categories in multiclass detection settings.

Keywords: code smells; representation learning; source code embeddings; CodeBERT; multiclass classification; software quality; software maintenance

1. Introduction

Code smells are indicators of potential deficiencies in software design or implementation that suggest the need for refactoring [1]. Reis et al. [2] investigated code smell detection based on collective knowledge by training supervised models on labels assigned by multiple teams for three code smells: Long Method, God Class, and Feature Envy. In their experimental setting, each code smell was formulated as an independent binary classification problem, and traditional classifiers were evaluated using cross-validation, with the ROC curve adopted as the primary performance metric. This approach enabled the analysis of specialized detectors for each individual code smell.

However, code smells are not necessarily mutually exclusive. Empirical studies on code smell co-occurrence have shown that multiple smells may affect the same software component. Palomba et al. [3] reported a high prevalence of smell co-occurrences in real-world systems and observed that nearly 59% of smelly classes contain more than one type of code smell. Similarly, Oizumi et al. [4] defined a smell agglomeration as a group of interrelated code smells affecting the same program

 doi:10.20944/preprints202603.1606.v1

location, and indicated that it is characterized by the co-occurrence of two or more smells within the same method, class, hierarchy, or package. Likewise, Santana et al. [5] examined combinations of code smells and found that Feature Envy consistently appears as a consequent in the identified association rules, suggesting the presence of non-trivial dependencies among code smells.

In this context, a binary formulation for each code smell may lead to a heterogeneous negative class, as a fragment that does not belong to the positive class of a given detector may nevertheless exhibit other smells. This situation introduces ambiguity in the negative class and may affect the reliability of the learned decision boundaries. Consequently, modeling code smell detection as a multiclass classification problem may provide a more coherent representation of the relationships among smell categories, especially in scenarios where different smells may coexist within the same code element.

Recent advances in representation learning have opened new possibilities for the automatic analysis of source code. Transformer-based models and neural embeddings are capable of capturing syntactic and semantic properties of programs directly from raw code, reducing the dependence on manually engineered metrics [6,7]. In this context, several studies have explored embedding-based representations for software engineering tasks such as vulnerability detection, defect prediction, and code summarization [8,9]. However, the extent to which these representations can effectively discriminate between different categories of code smells remains insufficiently explored, particularly under multiclass formulations.

Attentionsmelling evaluates GPT-4o in a prompt-based pipeline using an adapted oracle and reports that specialized smell descriptions and the inclusion of code metrics substantially improve LLM-based detection performance [10]. In contrast, the present study builds on the same empirical basis but adapts it into a different analytical pipeline centered on fixed vector representations and multiclass discriminative learning. Thus, while Attentionsmelling examines the effect of prompt engineering and contextual information on LLM-based smell identification, the present study investigates how embedding choice influences class separability and multiclass prediction.

From an experimental perspective, comparing code-specific and general-purpose embeddings is also methodologically relevant. While transformer-based models such as CodeBERT are explicitly pretrained on source code and thus embed domain-specific inductive biases, API-based general-purpose embeddings may still capture distributional regularities of program text without being specialized for software engineering tasks. Evaluating both types of representations makes it possible to examine whether code-specific pretraining provides a measurable advantage for multiclass code smell discrimination, or whether more general embedding spaces can achieve competitive performance.

The main contribution of this study is to provide empirical evidence on the effect of embedding choice on the separation of code smell categories within a multiclass formulation. In particular, the study compares transformer-based and API-based embedding representations and analyzes their impact on classification performance and on the spatial organization of code smells in the representation space.

Within this framework, the study addresses the following research questions:

- **RQ1.** Which of the evaluated embeddings provides the best separation among the three classes considered?
- **RQ2.** Is the observed performance difference consistent when classification results are analyzed separately for each code smell category?

2. Related Works

Automatic code smell detection has been addressed through approaches ranging from supervised models built on static code metrics to more recent methods based on neural embeddings and deep architectures. In general, the recent literature shows a progressive shift from manually engineered descriptors toward more expressive representations, as well as a growing interest in analyzing the

 doi:10.20944/preprints202603.1606.v1

effect of preprocessing, model optimization, and code representation on the final performance of the detectors.

2.1. Code Smell Detection Based on Metrics and Supervised Models

A considerable body of literature has addressed code smell detection using supervised models based on code metrics. Pecorelli et al. [11] compared heuristic approaches with machine learning models for metric-based detection, showing that supervised models can outperform heuristic detectors when adequate labeled data are available. This comparison is relevant because it reflects a shift in how the problem has been approached in the literature: detection is no longer viewed solely as the application of thresholds to metrics, but rather as a classification task that depends on both the model and the quality of the representation.

Other studies have explored specific variants within this framework. Dewangan and Rao [12] investigated method-level code smell detection using supervised models, whereas Putro et al. [13] proposed the Relevance Vector Machine as a probabilistic alternative for this task. Nandini et al. [14] examined how data balancing and parameter optimization can improve detection accuracy, while Dewangan et al. [15] evaluated ensemble learning algorithms for code smell detection. Taken together, these studies indicate that manually engineered feature-based approaches remain relevant and competitive, particularly when the goal is to develop interpretable detectors based on structural code descriptors.

2.2. Preprocessing, Feature Selection, and Pipeline Optimization

Recent literature has paid particular attention to the effect of preprocessing and pre-training stages on the performance of code smell detectors. In a systematic review, Santos and Choren [16] highlighted that tasks such as class balancing, normalization, feature selection, and other prior transformations substantially influence the performance of machine learning-based models. Complementarily, Zhang et al. [17] conducted a broad empirical study on feature selection and dimensionality reduction techniques, concluding that the impact of these decisions depends on both the code smell under analysis and the model employed.

These findings suggest that improvements in detection cannot be attributed solely to the classifier employed. Rather, the way in which information is filtered, transformed, and represented prior to training constitutes a central factor in the final performance of the system. From this perspective, the quality of the experimental pipeline becomes as relevant as the choice of the classification algorithm.

2.3. Embeddings and Deep Learning-Based Approaches

In contrast to metric-based approaches, several studies have explored distributional code representations and deep learning models. Kovačević et al. [18] proposed the automatic detection of Long Method and God Class using neural source code embeddings, showing that this type of representation can capture useful code properties without relying exclusively on manually engineered metrics. Subsequently, Škipina et al. [19] extended this line of research to the detection of Feature Envy and Data Class, further supporting the hypothesis that embeddings can model code relationships that are difficult to express solely through static descriptors.

Interest in representation is also reflected in comparative studies. Thakur et al. [20] evaluated different deep learning embedding techniques for code smell detection, showing that the choice of representation substantially affects the model's ability to discriminate among categories. Along a related line, Zhang et al. [21] combined pretraining and stacking models to improve generalization capability, whereas Ali et al. [22] explored a transformer-based approach to enhance the automatic detection of code smells.

Attentionsmelling represents a recent LLM-based approach to code smell analysis [10]. Its experimental design progressively incorporates prompt refinement, domain-specific smell descriptions, structured code metrics, and hyperparameter tuning, framing smell detection as an inference pipeline centered on a large language model rather than on conventional discriminative classification. This

 doi:10.20944/preprints202603.1606.v1

methodological perspective is relevant to the present study because it provides a contrasting pipeline for analyzing the same empirical setting..

In parallel, other studies have explored architectures aimed at capturing more complex dependencies in source code. Wang et al. [23] proposed a Bi-LSTM model with a self-attention mechanism for Feature Envy detection, with the aim of representing semantic dependencies relevant to this smell. Mesbah et al. [24] examined the applicability of Graph Neural Networks to code smell detection, highlighting the value of structural code relationships as a source of discriminative information. In a complementary direction, Thakur et al. [25] proposed COSTAR, an abstract tree-based representation designed to strengthen the structural encoding of source code. Taken together, these studies show a clear transition from metric-based approaches toward learned representations with greater syntactic, semantic, and relational richness.

3. Materials and Methods

Figure 1 illustrates the methodological workflow followed in this study, from dataset integration through preprocessing and embedding generation to model classification and evaluation.

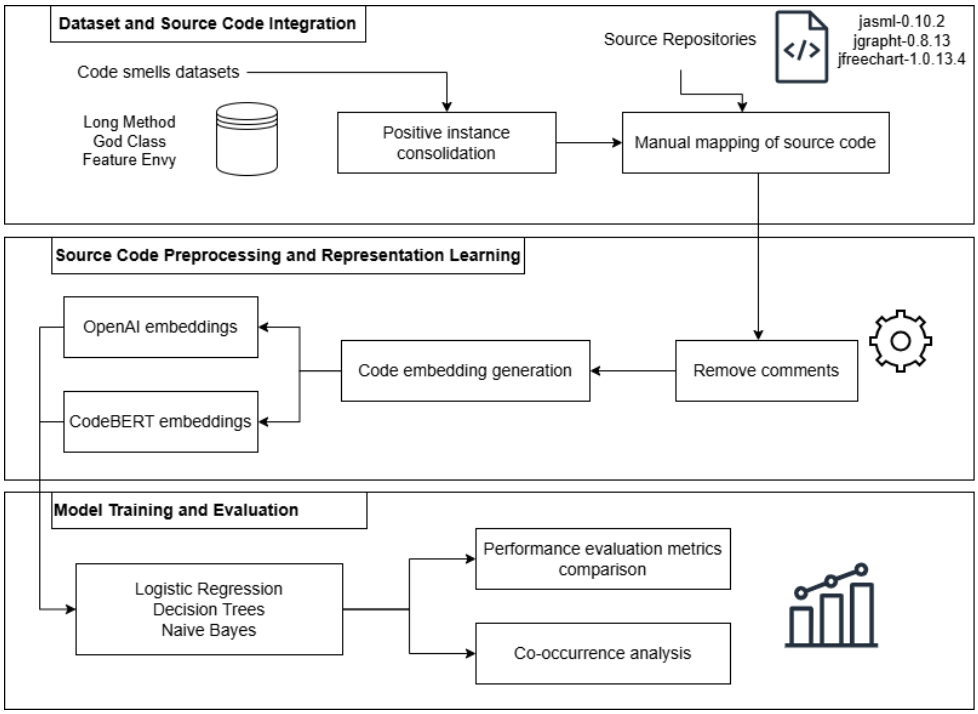


Figure 1. Method.

3.1. Dataset and Source Code Integration

The experimental dataset used in this study originates from the Crowdsmelling dataset proposed by Reis et al. [2]. In their work, the authors investigated the feasibility of detecting code smells through collective knowledge by aggregating labels provided by multiple development teams. The dataset contains instances of three widely studied code smells: Long Method, God Class, and Feature Envy, extracted from several open-source Java projects. In the original study, the detection task was formulated as three independent binary classification problems, and traditional machine learning classifiers were trained using code metrics derived from static analysis.

The present study builds upon the same empirical basis but adopts a different analytical perspective. Instead of treating each smell independently, the detection problem is reformulated as a multiclass classification task and relies on learned vector representations of source code rather than manually engineered metrics. This methodological shift allows the investigation of how different embedding models organize code smell categories within their latent representation spaces. In order to enable this analysis, the instances provided in the Crowdsmelling dataset were consolidated and linked to their

 doi:10.20944/preprints202603.1606.v1

corresponding source code fragments, allowing the generation of vector representations directly from the code rather than from manually engineered metrics. For the development of the experiment, the instances provided in the Crowdsmelling dataset were consolidated into a single multiclass dataset containing the three target smells: Long Method, God Class, and Feature Envy. This dataset was constructed from three code smells identified in Java code from the projects jasml-0.10.2, jgrapht-0.8.13, and jfreechart-1.0.13.4. From this dataset, only the instances labeled as true for the categories Long Method, God Class, and Feature Envy were consolidated. Each record contains information about the project, package, class, and method under analysis, together with a set of code metrics and its corresponding classification label. Table 1 presents the distribution of instances by class.

Table 1. Code Smell Instances.

Code Smell	Instances
Long Method	554
Feature Envy	191
God Class	166
Total	**911**

Because the dataset used in the experiment does not include the source code of the instances, the corresponding versions of the analyzed projects were downloaded, and a manual mapping was performed between each instance and its source code fragment. For this purpose, the project, package, class, and, when applicable, method information reported in each dataset record was used. In the case of God Class, the code was extracted at the class level starting from its declaration; therefore, import declarations were not included, whereas for Long Method and Feature Envy the extraction was performed at the method level. In this way, each instance was linked to its corresponding code fragment, resulting in the final dataset used in the subsequent analysis.

3.2. Source Code Preprocessing and Representation Learning

In this phase, line comments and block comments were removed in order to prevent the embeddings from incorporating textual information or non-executable content irrelevant to code smell classification [26]. In addition, line breaks were preserved to retain the structure of the code and the delimitation of its blocks [27].

To generate vector representations of the preprocessed source code, two embedding approaches were employed. In the first approach, the text-embedding-3-small model was accessed through the OpenAI API (OpenAI, San Francisco, CA, USA), producing 1536-dimensional embeddings. In the second approach, the pretrained microsoft/codebert-base model (Microsoft, Redmond, WA, USA) was implemented using the Transformers library (Hugging Face Inc., New York, NY, USA), producing 768-dimensional embeddings derived from the last hidden layer via mean pooling over valid tokens, followed by L2 normalization.

The comparison between OpenAI and CodeBERT was intentionally designed to contrast two distinct representation paradigms. CodeBERT is explicitly pretrained on programming language artifacts and natural language, which makes it a code-specialized embedding model with inductive biases oriented toward software structure and semantics. By contrast, OpenAI embeddings can be viewed as general-purpose distributional representations that are not specifically optimized for source code analysis or code smell discrimination. The goal of this comparison is therefore not to assume that both models are equivalent in their pretraining objectives, but rather to examine whether such specialization translates into a measurable advantage in multiclass code smell detection. In this sense, the comparison serves a dual purpose: first, as a predictive benchmark between two embedding paradigms; and second, as an empirical analysis of how code-specialized and general-purpose representations differ in their ability to organize smell categories in the latent space.

 doi:10.20944/preprints202603.1606.v1

3.3. Model Training and Evaluation

Training and evaluation were conducted using Logistic Regression, Decision Trees, and Naive Bayes for the multiclass classification of code smells based on the previously generated vector representations. To prevent information leakage between the training and test sets, the preprocessed source code was used as the grouping criterion, so that instances sharing the same preprocessed source code were not assigned to different partitions.

On this basis, evaluation was performed using five-fold stratified cross-validation with StratifiedGroupKFold. This scheme was chosen in view of the distribution of instances across classes, since a larger number of folds could reduce the representation of some code smells in particular partitions and compromise the stability of the evaluation. In each split, a pipeline composed of StandardScaler and LogisticRegression (max_iter = 4000, class_weight = balanced) was implemented using the scikit-learn library in order to standardize the input variables and mitigate the effect of class imbalance. Performance was assessed using macro F1-score, balanced accuracy, and the Matthews correlation coefficient (MCC), complemented by class-wise metrics and aggregated confusion matrices to analyze model behavior for each type of code smell.

As a complementary step to the predictive evaluation, a co-occurrence analysis was performed using grouped frequency counts to identify recurrent combinations of code smells in the dataset. Specifically, the analysis examined both repeated occurrences of the same smell type and joint occurrences of different smell types within the same source-code context. These frequency patterns were used as a descriptive complement to the classification results, particularly to explore whether recurrent co-occurrences were related to class overlap or to misclassification patterns observed in the confusion matrices.

4. Results

For the multiclass classification task, three machine learning classifiers were initially evaluated using embeddings generated by CodeBERT and OpenAI: Logistic Regression, Decision Tree, and Naive Bayes. As shown in Table 2, Logistic Regression with CodeBERT embeddings achieved the highest values for accuracy (0.9121), balanced accuracy (0.8929), F1-macro (0.8619), and ROC-AUC-OVR-macro (0.9880). Logistic Regression with OpenAI embeddings ranked second across these metrics, whereas Decision Tree and Naive Bayes showed lower performance for both embedding sources. Based on these results, Logistic Regression was selected as the final classification model.

Table 2. Performance comparison of the evaluated machine learning models for the multiclass classification task using CodeBERT and OpenAI embeddings.

Model	Embedding Source	Accuracy	Balanced Accuracy	F1-macro	ROC-AUC OVR macro
Logistic Regression	CodeBERT	**0.9121**	**0.8929**	**0.8619**	**0.9880**
	OpenAI	0.8159	0.7915	0.7622	0.9337
Decision Tree	CodeBERT	0.7556	0.6702	0.6151	0.7534
	OpenAI	0.5961	0.6109	0.5508	0.6919
Naive Bayes	CodeBERT	0.7368	0.5093	0.4563	0.7602
	OpenAI	0.7060	0.4904	0.4431	0.6617

4.1. Overall Classification Performance

Table 3 summarizes the overall classification performance obtained with the vector representations generated by OpenAI and CodeBERT under five-fold stratified group cross-validation. Across all primary evaluation metrics, CodeBERT outperformed OpenAI. Specifically, CodeBERT achieved a balanced accuracy of 0.8929 ± 0.1969, an F1-score macro of 0.8619 ± 0.1737, and an MCC of 0.7940 ± 0.2917, whereas OpenAI obtained 0.7915 ± 0.2192, 0.7622 ± 0.2251, and 0.6079 ± 0.3479, respectively.

 doi:10.20944/preprints202603.1606.v1

These results indicate that the representations derived from CodeBERT provided a more effective basis for the multiclass classification of the analyzed code smells.

Table 3. Overall classification performance under five-fold stratified group cross-validation.

Model	Balanced Accuracy	F1-Score Macro	MCC
OpenAI	0.7915 ± 0.2192	0.7622 ± 0.2251	0.6079 ± 0.3479
CodeBERT	0.8929 ± 0.1969	0.8619 ± 0.1737	0.7940 ± 0.2917

The observed standard deviations indicate that the classification performance was not uniform across folds for either representation approach. These global results provide an initial indication that the CodeBERT-based representations were more suitable for the classification task, a pattern that is further examined in the class-wise analysis and confusion matrix results.

4.2. Class-Wise Performance Analysis

Table 4 summarizes the class-wise performance obtained with the vector representations generated by OpenAI and CodeBERT in terms of precision, recall, and F1-score for each code smell category.

Table 4. Class-wise performance analysis.

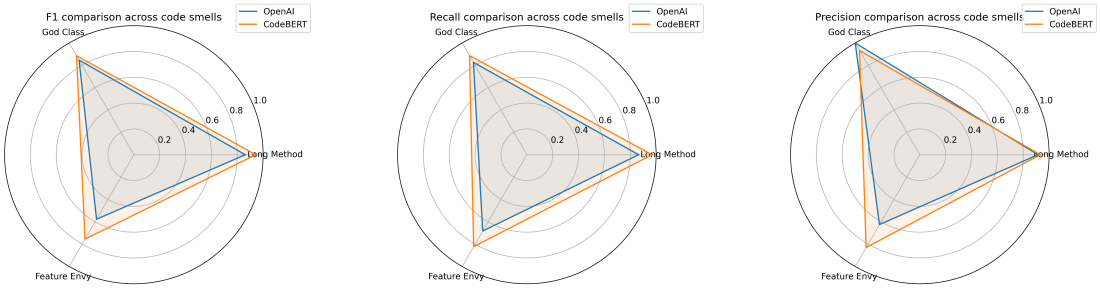
Model	Code Smell	Precision	Recall	F1-Score
OpenAI	Long Method	0.9142	0.8650	0.8625
OpenAI	God Class	1.0000	0.8267	0.8471
OpenAI	Feature Envy	0.6248	0.6829	0.5770
CodeBERT	Long Method	0.9319	0.9694	0.9439
CodeBERT	God Class	0.9333	0.8867	0.8864
CodeBERT	Feature Envy	0.8307	0.8227	0.7552

In both representation approaches, Long Method showed the highest F1-score, whereas Feature Envy presented the lowest values. For OpenAI, Long Method reached an F1-score of 0.8625, followed by God Class with 0.8471 and Feature Envy with 0.5770. For CodeBERT, Long Method again obtained the highest F1-score, with 0.9439, followed by God Class with 0.8864 and Feature Envy with 0.7552.

When comparing both approaches, CodeBERT achieved higher recall and F1-score for all three code smell categories. The largest difference was observed for Feature Envy, whose precision increased from 0.6248 to 0.8307, recall from 0.6829 to 0.8227, and F1-score from 0.5770 to 0.7552. For Long Method, the main increase was observed in recall, from 0.8650 to 0.9694, together with an F1-score increase from 0.8625 to 0.9439. For God Class, OpenAI obtained the highest precision value (1.0000), whereas CodeBERT showed higher recall and F1-score.

Figure 2 provides a visual comparison of precision, recall, and F1-score for each code smell category. The graphical representation reinforces the patterns observed in Table 4, particularly the stronger performance of Long Method in both approaches and the lower results obtained for Feature Envy. It also highlights the larger differences between OpenAI and CodeBERT in recall and F1-score for Feature Envy, as well as the higher recall and F1-score achieved by CodeBERT for God Class.

 doi:10.20944/preprints202603.1606.v1



(**a**) F1-score (**b**) Recall (**c**) Precision

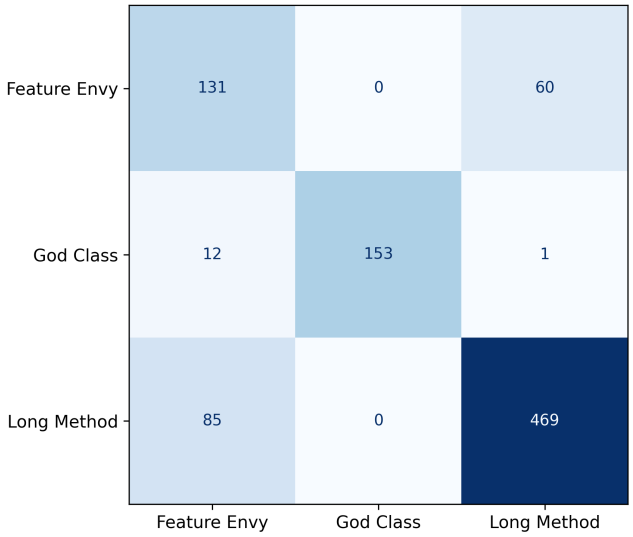
Figure 2. Comparison of evaluation metrics across models. (**a**) F1-score. (**b**) Recall. (**c**) Precision.

4.3. Confusion Matrix Analysis

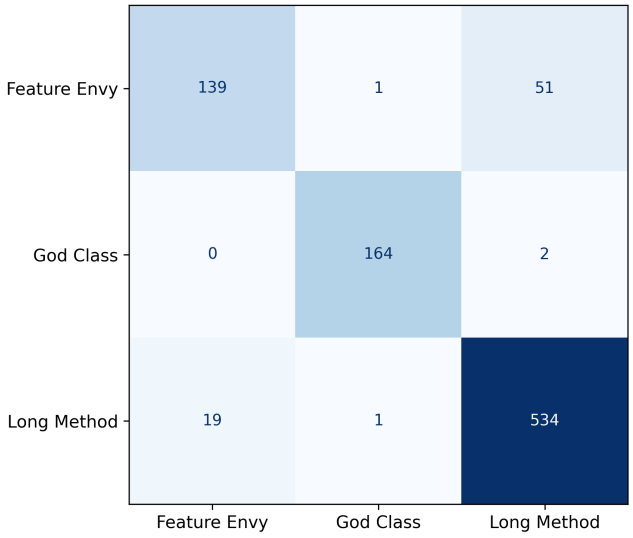
Figure 3 shows the aggregated confusion matrices for OpenAI and CodeBERT. In both cases, the main source of misclassification was observed between Feature Envy and Long Method. For OpenAI, 60 instances of Feature Envy were classified as Long Method, while 85 instances of Long Method were classified as Feature Envy. A smaller number of errors was observed for God Class, with 12 instances misclassified as Feature Envy and 1 instance misclassified as Long Method.

The confusion matrix obtained with CodeBERT shows a reduction in these errors. In particular, the number of Long Method instances misclassified as Feature Envy decreased from 85 to 19, and the number of Feature Envy instances misclassified as Long Method decreased from 60 to 51. For God Class, CodeBERT also produced fewer errors, with 164 correctly classified instances out of 166 and no misclassification into Feature Envy. These results are consistent with the class-wise metrics and show that the main performance gain of CodeBERT was associated with a clearer separation between Long Method and Feature Envy, while maintaining a high recognition rate for God Class.

 doi:10.20944/preprints202603.1606.v1



(**a**) OpenAI



(**b**) CodeBERT

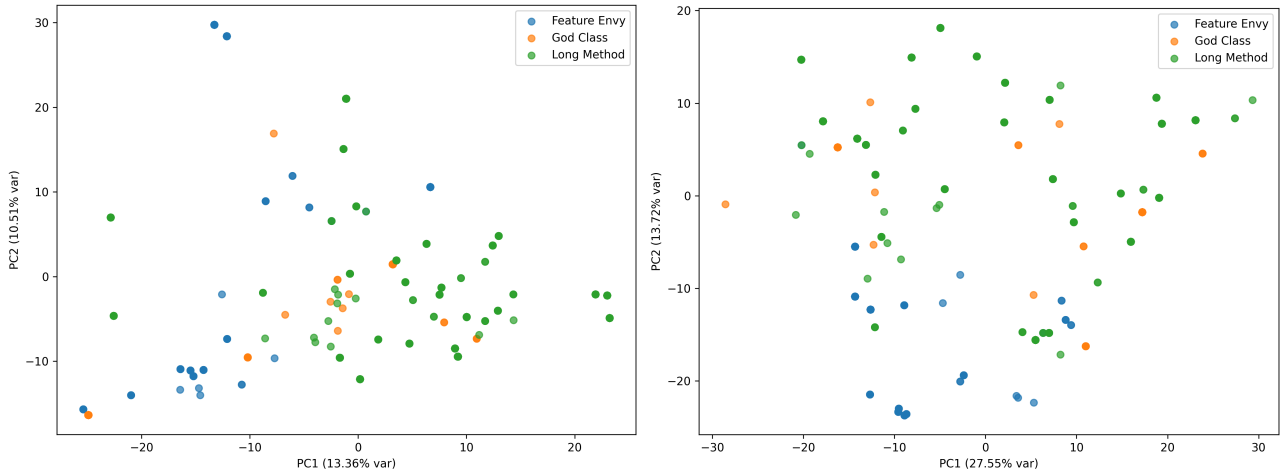
Figure 3. Confusion matrices obtained for the evaluated embedding models. (**a**) Confusion matrix for the OpenAI-based model. (**b**) Confusion matrix for the CodeBERT-based model.

4.4. Embedding Space Visualization

Figure 4 shows the two-dimensional PCA projections of the vector representations generated by OpenAI and CodeBERT. In the OpenAI projection, the first two principal components explained 13.36% and 10.51% of the variance, respectively. In the CodeBERT projection, the corresponding values were 27.55% and 13.72%, indicating that the first two components retained a larger proportion of the total variance.

In the OpenAI projection, Feature Envy was mainly concentrated in the left region of the plane, but still showed partial overlap with Long Method and God Class. Long Method was distributed more broadly across the central and right regions, whereas God Class occupied a more limited area without forming a fully isolated distribution. In contrast, the CodeBERT projection showed a clearer concentration of Feature Envy in the lower region of the plane, whereas Long Method was more frequently distributed in the upper and central regions. God Class remained located mainly between both groups, with partial overlap. Overall, the PCA visualization suggests a more structured spatial organization of the classes in the CodeBERT representation space than in the OpenAI projection.

 doi:10.20944/preprints202603.1606.v1



(**a**) OpenAI (**b**) CodeBERT

Figure 4. PCA projections of the embedding space. (**a**) PCA projection of OpenAI embeddings. (**b**) PCA projection of CodeBERT embeddings.

Figure 5 shows the cumulative explained variance obtained by PCA for the OpenAI and CodeBERT embeddings. In both models, the cumulative explained variance increased with the number of principal components. CodeBERT consistently showed higher cumulative explained variance values than OpenAI across the entire evaluated range. In particular, CodeBERT reached a cumulative explained variance of approximately 0.80 with nine principal components, whereas OpenAI required about sixteen components to reach a similar value.

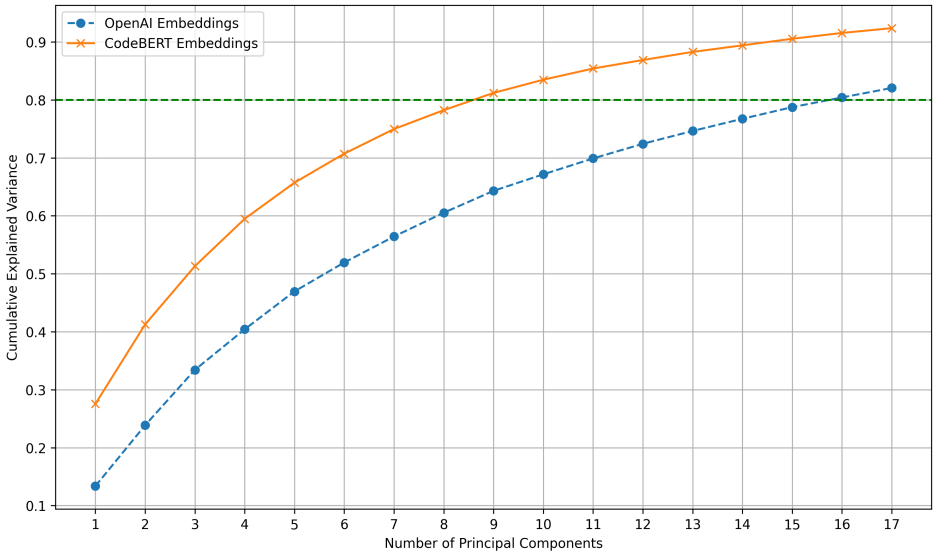


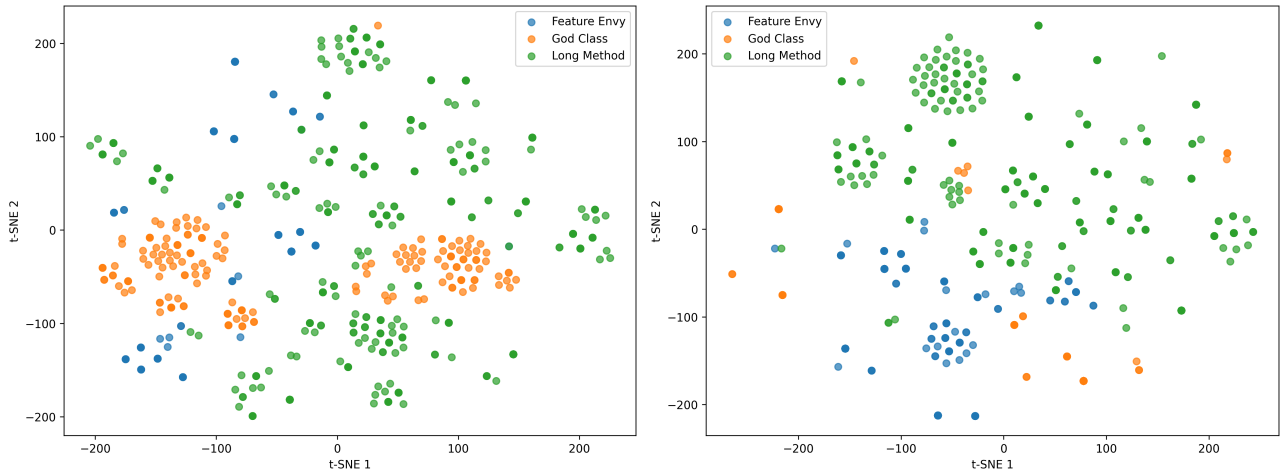
Figure 5. Cumulative explained variance.

Figure 6 shows the two-dimensional t-SNE projections for OpenAI and CodeBERT. In the OpenAI projection, Long Method was distributed over a broad portion of the space, particularly in the central and right regions, while God Class was concentrated mainly in compact groups located in the left and central-lower areas. Feature Envy appeared predominantly in the left and central portions of the plane, with partial overlap with God Class and a more limited overlap with Long Method.

In the CodeBERT projection, Feature Envy appeared more concentrated in the lower region of the plane, where several nearby local groups were observed. Long Method occupied a broader portion of the space, especially in the upper and central regions, whereas God Class appeared more sparsely distributed across the plane, with fewer compact groupings than in the OpenAI projection. Compared

 doi:10.20944/preprints202603.1606.v1

with OpenAI, the CodeBERT visualization showed a clearer spatial separation of Feature Envy from Long Method, although partial overlap among classes was still present in some regions.



(**a**) OpenAI (**b**) CodeBERT

Figure 6. t-SNE projections of the embedding spaces (**a**) t-SNE projection of OpenAI embeddings. (**b**) t-SNE projection of CodeBERT embeddings.

Overall, both visualizations showed a clearer spatial organization of the classes in the CodeBERT representation space than in the OpenAI projection.

4.5. Co-Occurrence Analysis

The co-occurrence analysis revealed non-uniform frequency patterns across the dataset. As shown in Table 5, the most frequent cross-type co-occurrence was God Class + Long Method, followed by God Class + Feature Envy, whereas Feature Envy + Long Method was rarely observed.

Table 5. Frequency of Cross-Type Code Smell Co-occurrences.

Code Smell Pair	Count
God Class + Long Method	34
God Class + Feature Envy	22
Feature Envy + Long Method	2

Table 6 further shows that these co-occurrences were concentrated in a limited set of recurring methods, while Table 7 indicates that same-type co-occurrences were also unevenly distributed across classes. Overall, these results suggest that code smell co-occurrences were concentrated in specific program locations rather than being evenly distributed throughout the dataset.

 doi:10.20944/preprints202603.1606.v1

Table 6. Methods Involved in Cross-Type Code Smell Co-occurrences.

Method	God Class	Long Method	Feature Envy
addMethodNameAndType	X	X	
dumpAttribute	X	X	
dumpConstantPool	X	X	
delComment	X	X	
parseClass	X	X	
parseOpcodes	X	X	
toString	X	X	
dumpClassHeader	X		X
dumpClassInfo	X		X
add	X		X
toString	X		X
parseClass		X	X
readAttribute		X	X

Table 7. Frequency of Same-Type Code Smell Co-occurrences by Class.

Class	Feature Envy	Long Method
ConstantPoolGenerator	12	-
JavaClassDumpper	2	2
Scanner	-	4
SourceCodeParser	-	16
JavaClassParser	6	5
SourceCodeBuilder	-	2
OpcodeLoader	-	2
Util	-	4
DateAxis	-	3

5. Discussion

The comparison among the three evaluated machine learning classifiers indicates that Logistic Regression was the most suitable model for the multiclass classification task. Across the selected evaluation metrics, it consistently achieved better performance than Decision Tree and Naive Bayes, which suggests that the decision boundaries required for this task were effectively captured by a linear classifier. This result also indicates that increasing classifier complexity did not necessarily lead to improved performance under the evaluated conditions. Therefore, Logistic Regression was retained as the reference classifier for the subsequent analysis.

Once Logistic Regression was selected as the reference classifier, an additional comparison was conducted between CodeBERT and OpenAI embeddings. The stronger results obtained with CodeBERT embeddings suggest that they provided a representation more closely aligned with the characteristics of the task. This may indicate that embeddings derived from a model specialized in source code captured more discriminative information than the more general-purpose OpenAI embeddings in this setting.

These results can be further interpreted in relation to Attentionsmelling as a closely related reference built on the same empirical basis [10]. However, the comparison should be understood at the methodological level rather than as a strict experimental replication, since Attentionsmelling evaluates a prompt-based GPT-4o pipeline, whereas the present study examines multiclass discriminative learning over fixed vector representations. Even under these differences, both studies suggest that code smell detection is strongly influenced by the quality of the information provided to the model. In Attentionsmelling, performance improves when the model receives refined prompts and structured code metrics, whereas in the present study performance improves when the classification pipeline relies on code-specialized embeddings rather than more general-purpose representations. At the class level, both studies also indicate that Feature Envy remains the most challenging category, reinforcing the view that this smell requires richer contextual information for reliable discrimination.

 doi:10.20944/preprints202603.1606.v1

The co-occurrence analysis provides additional context for interpreting the classification results. As shown in Table 5, the most frequent cross-type combinations were God Class + Long Method and God Class + Feature Envy, whereas Feature Envy + Long Method was rarely observed. However, the confusion matrix analysis showed that the main source of misclassification was precisely the pair Feature Envy–Long Method, and the class-wise results further indicated that Feature Envy was the most difficult category in both representation approaches. This contrast suggests that classification difficulty was not determined only by the explicit frequency of co-occurrence in the dataset, but also by structural similarities captured in the source code representations. In this sense, the present findings are consistent with prior evidence that code smell co-occurrences are a relevant phenomenon and that relationships between smell categories cannot be reduced to isolated occurrences alone [3]. From this perspective, the co-occurrence results complement the predictive evaluation by indicating that overlap among code smell categories may arise both from their joint presence in the same code context and from shared characteristics that reduce class separability in the embedding space.

Taken together, these findings allow the two research questions of this study to be addressed.

5.1. RQ1. Which of the Evaluated Embeddings Provides the Best Separation Among the Three Classes Considered?

The results indicate that CodeBERT provided the best separation among the three analyzed classes. This conclusion is supported by the overall classification results, where CodeBERT achieved higher balanced accuracy, macro F1-score, and MCC than OpenAI. The same pattern was observed in the confusion matrices, where CodeBERT reduced the number of misclassifications, particularly between Feature Envy and Long Method, which represented the main source of confusion in both approaches.

This difference can also be interpreted in light of the nature of the compared embedding models. Because CodeBERT is pretrained on source code corpora, it is more likely to encode structural and syntactic regularities that are directly relevant to code smell discrimination. In contrast, OpenAI embeddings, while still able to capture meaningful distributional patterns, are not explicitly optimized for software engineering tasks. The superiority of CodeBERT in the present study therefore suggests that domain-specific pretraining provides a more suitable latent space for separating smell categories, especially when the task requires distinguishing between structurally related smells such as Long Method and Feature Envy.

The dimensionality reduction analyses were consistent with these results. In the PCA projection, the CodeBERT embeddings showed a more structured spatial organization, with Feature Envy concentrated mainly in the lower region of the plane and Long Method more frequently located in the upper and central regions. In the t-SNE projection, the same general tendency was observed, with a clearer grouping of Feature Envy and a broader distribution of Long Method, while God Class remained between both classes with partial overlap. Although some overlap is still present, both visualizations suggest that CodeBERT generates a representation space with a clearer class organization than OpenAI.

A complementary insight emerges from the cumulative variance analysis of the PCA decomposition. While the two-dimensional projections shown in Figure 5 capture only a limited portion of the total variance, the number of components required to reach 80% of the explained variance differs substantially between the two embedding models. In particular, CodeBERT requires only nine principal components to reach this threshold, whereas the OpenAI embeddings require sixteen components.

This result suggests that CodeBERT organizes the information relevant to code smell discrimination in a more compact latent space. In other words, a smaller number of orthogonal directions is sufficient to capture most of the variability of the representation. From a machine learning perspective, this property facilitates the separation of classes using relatively simple classifiers, which is consistent with the superior performance observed for CodeBERT in the classification experiments. Conversely, the higher number of components required by the OpenAI embeddings indicates a more dispersed representation space, where discriminative information is spread across a larger number of dimensions.

 doi:10.20944/preprints202603.1606.v1

This observation is also consistent with the t-SNE visualizations presented in Figure X, where the CodeBERT embeddings exhibit clearer local grouping patterns among the code smell categories.

Additional insight can be obtained from the confusion matrix analysis presented in Section 4.3. The main source of misclassification for both embedding approaches occurs between Feature Envy and Long Method. This pattern suggests that these two smells share structural characteristics that make them difficult to distinguish using purely distributional representations of source code. Long methods frequently involve multiple interactions with external objects or classes, which may generate token patterns similar to those observed in Feature Envy instances.

Nevertheless, the confusion matrices also show that CodeBERT substantially reduces this ambiguity. In particular, the number of Long Method instances misclassified as Feature Envy decreases markedly when using CodeBERT embeddings. This result indicates that code-specialized representations capture structural relationships in the source code more effectively than general-purpose embeddings. In contrast, the God Class category exhibits relatively low confusion with the other smells, suggesting that its structural characteristics are more distinct and easier to capture within the embedding space. This observation is also consistent with the PCA and t-SNE visualizations, where partial overlap between Long Method and Feature Envy can be observed, particularly in the OpenAI embedding space.

Therefore, the results consistently indicate that CodeBERT provides the most effective separation among the three code smell classes considered in this study.

5.2. RQ2. Is This Difference Maintained When Performance Is Analyzed Separately for Each Code Smell?

The class-wise results indicate that the advantage of CodeBERT remains when performance is analyzed separately for each code smell. For Long Method, both approaches achieved strong results, but CodeBERT obtained higher recall and F1-score, indicating a more accurate identification of instances in this class. For God Class, OpenAI reached the highest precision, whereas CodeBERT achieved higher recall and F1-score, which reflects a more balanced performance. The largest difference was observed for Feature Envy, where CodeBERT improved precision, recall, and F1-score with respect to OpenAI.

These class-wise patterns are also consistent with the confusion matrix analysis. In particular, the largest reduction in errors with CodeBERT was observed in the confusion between Feature Envy and Long Method, which supports the improvement detected for Feature Envy in the class-wise metrics. This result is especially relevant because Feature Envy was the most challenging class in both approaches.

A possible explanation for these differences can be related to the structural characteristics of the analyzed code smells. Long Method and God Class are primarily associated with size-related properties of the code, such as excessive method length or large classes containing many responsibilities. These characteristics tend to generate relatively consistent structural patterns that can be captured by both embedding approaches. In contrast, Feature Envy is defined by an abnormal dependency structure in which a method relies excessively on the data of another class. This behavior may produce more subtle contextual patterns that are harder to capture using general-purpose embeddings. Consequently, the stronger performance of CodeBERT for Feature Envy suggests that code-specialized representations are better suited to modeling these structural relationships between program elements.

Therefore, the results confirm that the advantage of CodeBERT is consistently preserved across the individual code smell categories, although the magnitude of the improvement varies depending on the structural characteristics of each smell. The difference observed at the global level was preserved in the class-wise analysis, with CodeBERT showing better performance for the three code smell categories, although the improvement was more pronounced for Feature Envy than for Long Method and God Class.

 doi:10.20944/preprints202603.1606.v1

6. Conclusions

This study presented an empirical comparison of different embedding-based representations for the multiclass detection of code smells in source code. In contrast to previous studies that formulate smell detection as independent binary classification problems, the present work adopted a multiclass formulation that allows the simultaneous discrimination of three common smells: Long Method, God Class, and Feature Envy. Using the Crowdsmelling dataset as an empirical basis, source code fragments were extracted and transformed into vector representations through two embedding approaches: a general-purpose embedding model and the code-specialized CodeBERT model.

The experimental results consistently showed that CodeBERT provides superior performance across the evaluated metrics, including balanced accuracy, macro F1-score, and Matthews correlation coefficient. This advantage was also confirmed in the class-wise analysis, where CodeBERT achieved more balanced precision and recall values across all three smells. In particular, the improvement was most evident for the Feature Envy category, which represented the most challenging class in the dataset due to its structural similarity with Long Method.

This study also provides empirical evidence that embedding choice significantly influences the separability of code smell categories in multiclass detection settings.

The dimensionality reduction analyses further supported these findings. Both PCA and t-SNE visualizations suggested that CodeBERT organizes code smell instances in a more structured latent representation space. In addition, the cumulative variance analysis indicated that CodeBERT concentrates most of the representation variance in a smaller number of principal components compared to the general-purpose embeddings. This property suggests that domain-specific pretraining enables a more compact and semantically meaningful representation of program structure, which facilitates the separation of smell categories using relatively simple classifiers.

The confusion matrix analysis provided additional insight into the nature of the classification errors. The main source of confusion was observed between Feature Envy and Long Method, which is consistent with the structural relationships between these smells. Nevertheless, CodeBERT significantly reduced this ambiguity, indicating that code-specialized embeddings capture structural dependencies between program elements more effectively than general-purpose representations.

Despite these results, the study presents several limitations. First, the dataset size is relatively small and limited to three specific code smells, which may restrict the generalization of the findings to other smell categories or programming languages. Second, the evaluation focused on static code fragments without considering additional contextual information such as project evolution or software architecture.

Future work may extend this research in several directions. Larger datasets and additional code smells could be incorporated to evaluate the robustness of embedding-based approaches in more diverse scenarios. Furthermore, combining embedding representations with structural program analysis or graph-based models may provide richer representations of code dependencies. Finally, exploring explainability techniques for embedding-based smell detection models may offer further insights into how representation learning captures the structural characteristics of problematic code.

Author Contributions: Conceptualization, M.M. and R.B.; methodology, M.M. and R.B.; software, M.M.; validation, R.B.; formal analysis, M.M. and; investigation, M.M. and R.B.; data curation, M.M.; writing—original draft preparation, M.M.; writing—review and editing, M.M. and R.B.; visualization, M.M. and R.B.; supervision, R.B. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: The dataset analyzed in this study is publicly available. Crowdsmelling dataset was obtained from GitHub and can be accessed at: https://github.com/dataset-cs-surveys/Crowdsmelling. The source code, experimental notebooks, and processed artifacts supporting the reported results are publicly available at: https://github.com/marcela-mosquera/Evaluating-Embedding-Representations-for-Multiclass-Code-Smell-

 doi:10.20944/preprints202603.1606.v1

Detection. The processed dataset generated through dataset integration, manual source-code mapping, and preprocessing is publicly available at: https://doi.org/10.6084/m9.figshare.31717957.

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

References

1. Fowler, M. *Refactoring: Improving the Design of Existing Code*, 2nd ed.; Addison-Wesley Professional: 2018.
2. Reis, J. P.; Abreu, F. B. e.; Carneiro, G. F. Crowdsmelling: A Preliminary Study on Using Collective Knowledge in Code Smells Detection. *Empirical Software Engineering* **2022**, *27*, 69. DOI: 10.1007/s10664-021-10110-5.
3. 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. DOI: 10.1016/j.infsof.2018.02.004.
4. Oizumi, W.; Sousa, L.; Oliveira, A.; Garcia, A.; Agbachi, A. B.; Oliveira, R.; Lucena, C. On the Identification of Design Problems in Stinky Code: Experiences and Tool Support. *J. Braz. Comput. Soc.* **2018**, *24*, 13. DOI: 10.1186/s13173-018-0078-y.
5. Santana, A.; Cruz, D.; Figueiredo, E. An Exploratory Study on the Identification and Evaluation of Bad Smell Agglomerations. In Proceedings of the 36th Annual ACM Symposium on Applied Computing, Virtual Event, Republic of Korea, 22–26 March 2021; pp. 1289–1297. DOI: 10.1145/3412841.3442003.
6. Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "CodeBERT: A Pre-Trained Model for Programming and Natural Languages," in *Findings of the Association for Computational Linguistics: EMNLP 2020*, T. Cohn, Y. He, and Y. Liu, Eds. Association for Computational Linguistics, 2020, pp. 1536–1547. doi: 10.18653/v1/2020.findings-emnlp.139. Available: https://aclanthology.org/2020.findings-emnlp.139/
7. A. Kanade, P. Maniatis, G. Balakrishnan, and K. Shi, "Learning and Evaluating Contextual Embedding of Source Code," in *Proceedings of the 37th International Conference on Machine Learning (ICML 2020)*, H. Daumé III and A. Singh, Eds., vol. 119, *Proceedings of Machine Learning Research*, pp. 5110–5121, PMLR, 2020. Available: https://proceedings.mlr.press/v119/kanade20a.html
8. U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning Distributed Representations of Code," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, Art. no. 40, pp. 1–29, Jan. 2019. doi: 10.1145/3290353.
9. M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, "A Survey of Machine Learning for Big Code and Naturalness," *ACM Computing Surveys*, vol. 51, no. 4, Art. 81, pp. 1–37, Jul. 2018. doi: 10.1145/3212695.
10. Gomes, A.; Sousa, D.; Maia, P.; Paixao, M. Attentionsmelling: Using Large Language Models to Identify Code Smells. In Proceedings of the XXXIX Simpósio Brasileiro de Engenharia de Software, Recife/PE, Brazil, 2025; pp. 271–281. DOI: 10.5753/sbes.2025.9921.
11. Pecorelli, F.; Palomba, F.; Di Nucci, D.; De Lucia, A. Comparing Heuristic and Machine Learning Approaches for Metric-Based Code Smell Detection. In Proceedings of the IEEE International Conference on Program Comprehension, 2019; pp. 93–104. DOI: 10.1109/ICPC.2019.00023.
12. Dewangan, S.; Rao, R. S. Method-Level Code Smells Detection Using Machine Learning Models. In *Lecture Notes in Networks and Systems* **2023**, *725*, 77–86. DOI: 10.1007/978-981-99-3734-9_7.
13. Putro, H. P.; Yuhana, U. L.; Yuniarno, E. M.; Purnomo, M. H. Relevance Vector Machine for Code Smell Detection. In Proceedings of the IEEE International Conference on Industrial Informatics (INDIN), 2024. DOI: 10.1109/INDIN58382.2024.10774521.
14. Nandini, A.; Singh, R.; Rathee, A. Improving Machine Learning Algorithm's Accuracy for Detecting Code Smell Using Data Balancing and Parameter Optimization. In *Lecture Notes in Networks and Systems* **2025**, *1128*, 151–162. DOI: 10.1007/978-981-97-7371-8_12.
15. Dewangan, S.; Rao, R. S.; Mishra, A.; Gupta, M. Code Smell Detection Using Ensemble Machine Learning Algorithms. *Appl. Sci.* **2022**, *12*, 10321. DOI: 10.3390/app122010321.
16. Santos, F. do R.; Choren, R. Data Preprocessing for Machine Learning Based Code Smell Detection: A Systematic Literature Review. *Inf. Softw. Technol.* **2025**, *184*, 107752. DOI: 10.1016/j.infsof.2025.107752.
17. Zhang, Z.; Zhu, L.; Yin, S.; Hu, W.; Gao, S.; Chen, H.; Li, F. The Impact of Feature Selection and Feature Reduction Techniques for Code Smell Detection: A Comprehensive Empirical Study. *Autom. Softw. Eng.* **2025**, *32*, 2. DOI: 10.1007/s10515-025-00524-6.

 doi:10.20944/preprints202603.1606.v1

18. Kovačević, A.; Slivka, J.; Vidaković, D.; Grujić, K.-G.; Luburić, N.; Prokić, S.; Sladić, G. Automatic Detection of Long Method and God Class Code Smells through Neural Source Code Embeddings. *Expert Syst. Appl.* **2022**, *204*, 117607. DOI: 10.1016/j.eswa.2022.117607.
19. Škipina, M.; Slivka, J.; Luburić, N.; Kovačević, A. Automatic Detection of Feature Envy and Data Class Code Smells Using Machine Learning. *Expert Syst. Appl.* **2024**, *243*, 122855. DOI: 10.1016/j.eswa.2023.122855.
20. Thakur, P. S.; Jadeja, M.; Chouhan, S. S.; Rathore, S. S. Evaluating Deep Learning Embedding Techniques for Code Smell Detection. In *Lect. Notes Comput. Sci.* **2025**, *15526*, 339–350. DOI: 10.1007/978-3-031-81821-9_21.
21. Zhang, D.; Song, S.; Zhang, Y.; Liu, H.; Shen, G. Code Smell Detection Research Based on Pre-training and Stacking Models. *IEEE Lat. Am. Trans.* **2024**, *22*, 22–30. DOI: 10.1109/TLA.2024.10375735.
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. DOI: 10.3390/app15084559.
23. Wang, H.; Liu, J.; Kang, J.; Yin, W.; Sun, H.; Wang, H. Feature Envy Detection Based on Bi-LSTM with Self-Attention Mechanism. In Proceedings of the 2020 IEEE International Symposium on Parallel and Distributed Processing with Applications, 2020 IEEE International Conference on Big Data and Cloud Computing, 2020 IEEE International Symposium on Social Computing and Networking, and 2020 IEEE International Conference on Sustainable Computing and Communications, 2020; pp. 448–457. DOI: 10.1109/ISPA-BDCloud-SocialCom-SustainCom51426.2020.00082.
24. Mesbah, D.; El Madhoun, N.; Al Agha, K.; Chalouati, H. Beyond the Code: Unraveling the Applicability of Graph Neural Networks in Smell Detection. In *Lect. Notes Data Eng. Commun. Technol.* **2024**, *224*, 148–161. DOI: 10.1007/978-3-031-72325-4_15.
25. Thakur, P. S.; Jadeja, M.; Chouhan, S. S.; Rathore, S. S. COSTAR: Software Code Smell Detection Through Tree-Based Abstract Representation. *IEEE Trans. Reliab.* **2026**, *75*, 581–595. DOI: 10.1109/TR.2025.3648404.
26. Fudholi, D. R.; Capiluppi, A. Artificial Intelligence for Source Code Understanding Tasks: A Systematic Mapping Study. *Inf. Softw. Technol.* **2026**, *189*, 107915. DOI: 10.1016/j.infsof.2025.107915.
27. Wang, Z.; Li, G.; Li, J.; Dong, Y.; Xiong, Y.; Jin, Z. Line-Level Semantic Structure Learning for Code Vulnerability Detection. In Proceedings of the 16th International Conference on Internetware, 2025; pp. 269–280. DOI: 10.1145/3755881.3755894.