2025 20th International Joint Symposium on Artificial Intelligence and Natural Language Processing (iSAI-NLP) | 979-8-3315-0217-1/25/$31.00 ©2025 IEEE | DOI: 10.1109/ISAI-NLP66160.2025.11320518

Code Smell Classification Using Graph Convolutional Network with Imbalanced Data and Model Integration

Phawinee Suphawimon
School of Information Technology
King Mongkut's Institute of Technology Ladkrabang
Bangkok, Thailand
67076047@kmitl.ac.th

Tuchsanai Ploysuwan
School of Information Technology
King Mongkut's Institute of Technology Ladkrabang
Bangkok, Thailand
tuchsanai@it.kmitl.ac.th

***Abstract*—Code smell represents critical design anomalies that significantly impact software maintainability and quality. This paper presents a comprehensive framework using Graph Convolutional Networks (GCNs) integrated with traditional machine learning techniques. We systematically evaluated graph construction approaches, model integration methodologies, and data balancing strategies using nine real-world Python repositories labeled with PyExamine. Our methodology combines BERT embeddings with graph structural representations, implementing layer integration (Method I) and feature concatenation (Method II). Results show per-line graph construction outperforms global approaches, with SMOTE achieving 96.14% accuracy compared to 86.70% for imbalanced data. Including non-smelly code improves performance from 71% to 95%, demonstrating the importance of negative examples. Our ablation study shows explicit feature engineering achieves only 67% accuracy compared to 95% for end-to-end learning. The integrated GCN with Transformer using Method II achieved 95% accuracy and 89% F1-score, nearly matching CodeT5 (97% accuracy, 85% F1-score) while providing better interpretability.**

***Index Terms*—code smell classification, graph convolutional network, model integration, imbalanced data handling, Python code analysis**

I. Introduction

Code smells represent design problems that indicate deeper issues in software architecture, making code difficult to understand, modify, and extend [8]. Their identification and remediation are crucial for maintaining software quality and reducing technical debt. Traditional rule-based approaches suffer from high false positive rates and limited adaptability, while machine learning techniques have shown promise but face challenges with imbalanced datasets and complex structural relationships in code.

Machine learning techniques for code smell detection have addressed tool subjectivity issues [9]. The emergence of deep learning and graph neural networks has opened new possibilities for capturing complex structural relationships, with Graph Convolutional Networks (GCNs) demonstrating significant success through message passing between graph nodes [10].

A. Research Questions

RQ1: What is the most effective graph construction strategy for code smell classification: global vocabulary graphs or per-line graphs?

RQ2: How does the performance of standalone GCN models compare with integrated GCN approaches combined with traditional machine learning models?

RQ3: Which integration methodology (Method I vs. Method II) provides superior performance for combining GCN with traditional models?

RQ4: How do GCN-integrated models perform compared to specialized Large Language Models for code smell detection?

II. Related Work

A. Code Smell Classification Approaches

1) Traditional Approaches: Early code smell detection approaches relied primarily on rule-based systems and metric thresholds. These methods suffer from high false positive rates and limited adaptability across different software contexts. The subjective nature of threshold selection has been identified as a major limitation [12].

2) Machine Learning-Based Detection: Supervised learning-based code smell detection has become a dominant approach [7]. Recent surveys show growing adoption of machine learning techniques including decision trees, support vector machines, random forests, and neural networks [6]. Python code smell detection using conventional machine learning models has been explored with datasets focusing on Large Class and Long Method code smells [11].

3) Deep Learning Approaches: There has been growing interest in utilizing deep learning techniques for code smell detection, as code smells are indicators of deeper problems in source code that affect system maintainability and evolution [14]. The DeleSmell framework demonstrates effectiveness of combining deep learning with latent semantic analysis [6].

B. Graph Neural Networks in Software Engineering

Graph Neural Networks are specially crafted to process graph-structured data with exceptional capability to grasp intri-

cate relationships [15]. GCNs apply convolution operations to graph-structured data [16]. Recent surveys provide comprehensive overviews of GCN-based approaches for text classification [13]. Zhang et al. pioneered GNN-based approaches for code smell classification using document-level graphs with BERT embeddings [5].

C. Imbalanced Data Handling Techniques

SMOTE addresses class imbalance by generating synthetic minority samples through interpolation between existing minority instances [17], though it may magnify data complexity [18]. Genetic Algorithm-based over-sampling approaches optimize synthetic sample generation by evolving populations of candidate solutions [2].

D. Integration of Graph and Traditional Models

VGCN-BERT demonstrates successful integration of graph embeddings with transformer models for text classification [3]. Recent frameworks like Smell-ML [1] address detection of rarely studied code smells, while PyExamine provides comprehensive smell detection capabilities with 49 distinct metrics [4].

III. Methodology

A. Problem Formulation

We formalize code smell classification as a multi-class classification problem. Given a code element c, predict its label $y \in \{0, \ldots, 10\}$ representing 11 classes: Feature Envy, Large Class, Long Method, Long Parameter List, Message Chains, Non-smell, Potential Divergent Change, Potential Shotgun Surgery, Speculative Generality, Switch Statements, and Temporary Field.

B. Dataset and Preprocessing

The dataset comprises 9 open-source Python projects from GitHub: dmtpy, MELODI, oteapi-pipelines, pseudo-hamiltonian-neural-networks, rewts, RTT_for_APR, simapy, sisana, and sponge. Code smell identification was performed using PyExamine [4].

C. Code Representation and Graph Construction Strategies

We evaluated two fundamental approaches:

Approach 1: Global Vocabulary Graph constructs a single graph from the entire training dataset, captures global vocabulary relationships and semantic patterns, uses BERT embeddings for semantic similarity computation, and creates edges based on co-occurrence patterns across the dataset.

Approach 2: Per-Line Graph Construction builds individual graphs for each line of code, preserves local structural information within code segments, enables consistent graph structure between training and inference, and better handles unseen data by maintaining structural consistency.

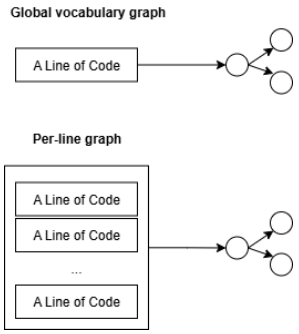

Fig. 1: Comparison between global vocabulary graph (Approach 1) and per-line graph construction (Approach 2). Per-line graphs maintain consistent structure between training and inference phases.

D. Code Representation Using BERT Embeddings

BERT provides contextual embeddings that effectively capture semantic relationships between different code segments. The BERT embeddings serve dual purposes: (1) **Node Features:** Each code line is represented as a BERT embedding vector, and (2) **Edge Weights:** Semantic similarity between code segments is computed using cosine similarity between BERT embeddings.

E. Graph Convolutional Network Architecture

The GCN architecture employs a two-layer framework for node representation learning. The first convolutional layer transforms BERT embeddings to a hidden representation space and the second layer produces final embeddings. The model incorporates batch normalization after the initial convolutional layer and dropout regularization ($p = 0.2$).

F. Model Integration Framework

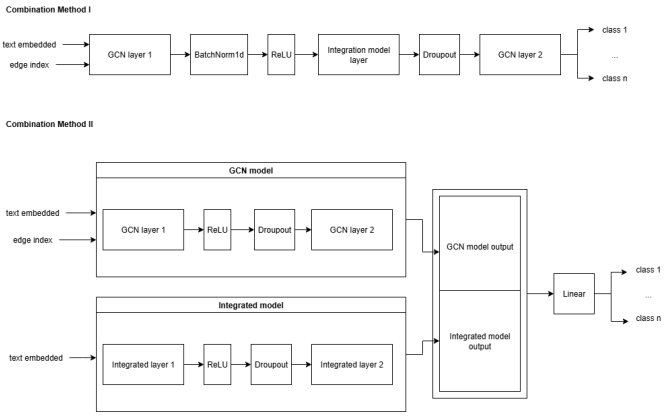

Fig. 2: Integration model structure comparison between Method I (layer integration within GCN architecture) and Method II (feature concatenation approach).

We evaluate two integration methodologies by combining GCN with various baseline models including neural networks

(MLP, CNN, RNN, LSTM, GRU, Transformer, BERT) and traditional ML models (DT, LR, RF, SVM).

Method I: Integration Model as Layer within GCN Architecture - The integration model is embedded as an intermediate layer within the GCN architecture, creating a unified end-to-end learning framework.

Method II: GCN as Feature Engineering for Integration Model - This method treats the GCN model as a sophisticated feature engineering component that generates high-level graph representations. The GCN output (enhanced BERT embeddings) is concatenated with traditional model output for final prediction.

G. Imbalanced Data Handling Strategies

We implement three approaches: (1) **Original Data:** Baseline using natural class distribution with class weighted loss functions, (2) **SMOTE:** Synthetic Minority Oversampling Technique generates synthetic minority samples using k-nearest neighbors ($k = 5$), and (3) **Genetic Algorithm:** Evolutionary optimization using original data as initial population, fitness function evaluation, parent selection, crossover for characteristic exchange, mutation with randomization, and survival selection.

H. Evaluation Metrics

Given the imbalanced nature of code smell datasets, we employ **Accuracy** (overall proportion of correctly classified instances) and **Macro-averaged F1-score** (unweighted average treating each class equally).

IV. Experimental Setup

Data was split into training (70%), validation (15%), and testing (15%) sets. K-fold cross-validation was implemented with k ranging from 2 to 9, selecting the optimal k based on lowest mean final training loss. Grid search was employed for hyperparameter optimization of traditional ML models.

The model utilizes BERT embeddings as the sole input features. Semantic similarity between code segments is computed using cosine similarity of their BERT embeddings to determine edge weights. StandardScaler from scikit-learn is applied to normalize the BERT embedding dimensions.

V. Results and Analysis

A. RQ1: Graph Construction Strategy Comparison

Global graph approach showed excellent training performance but poor generalization. Model learned global vocabulary relationships that couldn't be maintained when creating separate graphs for unseen data, causing significant performance degradation on inference.

Per-line graph construction demonstrated consistent performance across both training and inference phases with comparable and reliable performance. Better handling of unseen data due to consistent graph structure. Model captures diverse patterns at code line level, making it robust to variations.

Conclusion for RQ1: Per-line graph construction is superior for code smell detection, providing better generalization capabilities and consistent performance across training and inference phases.

B. RQ2 & RQ3: Model Integration Results

Using SMOTE-balanced data with per-line graphs and including non-smell data as label 0:

1) Ablation Study: Feature Engineering Approach: We initially explored a two-stage feature engineering approach where code structure type classification serves as an intermediate feature extraction step before code smell detection. Fig. 3 presents the performance comparison. Results demonstrate that incorporating explicit feature engineering steps degrades performance across all model combinations:

Feature Engineering Approach: Best accuracy 67% (GCN_CNN Method I, GCN_BERT Method I), best F1-score 67% (GCN_RNN Method I, GCN_BERT Method I).

Direct Classification Approach: Best accuracy 95% (Table III), best F1-score 89% (GCN_Transformer Method II).

This significant performance gap (67% vs. 95% accuracy) indicates that intermediate feature engineering introduces information bottleneck. The GCN with BERT embeddings can effectively capture code structural relationships without explicit categorical feature extraction. All subsequent experiments employ the direct classification approach.

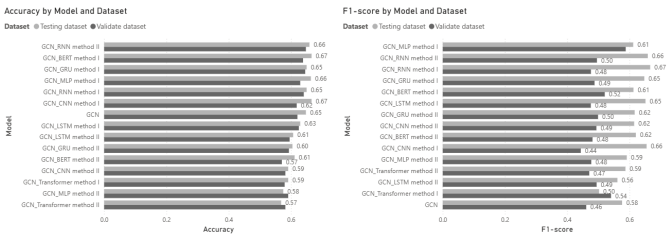

Fig. 3: Performance comparison of feature engineering approach. The feature engineering approach achieves significantly lower performance (max 67% accuracy) compared to direct classification (95% accuracy in Table III).

TABLE I: ABLATION STUDY: IMBALANCED DATA HANDLING

Method	Test Accuracy	Test F1-Score
Original Data	0.87	0.90
SMOTE	**0.96**	**0.96**
Genetic Algorithm	0.38	0.42

2) Class Distribution Strategy: Smelly-Only vs Combined Data: A critical design decision in code smell detection is whether to train models exclusively on smelly code samples or to include non-smelly code as a separate class. We systematically evaluated both strategies to determine the optimal class distribution approach.

Dataset Configuration:

Smelly-Only Dataset: Contains only code samples with identified smell types (10 smell classes: Feature Envy, Large

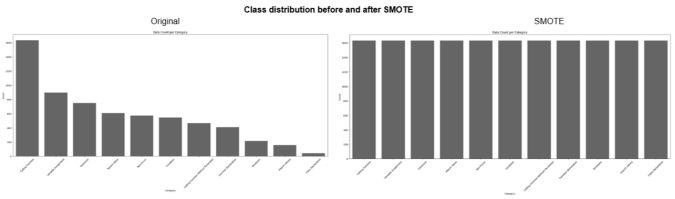

Fig. 4: Class distribution of code smell types before and after SMOTE oversampling. SMOTE effectively balances minority classes by generating synthetic samples.

Class, Long Method, Long Parameter List, Message Chains, Potential Divergent Change, Potential Shotgun Surgery, Speculative Generality, Switch Statements, and Temporary Field). This configuration assumes that smell detection operates on pre-filtered suspicious code segments.

Combined Dataset: Includes both smelly and non-smelly code samples (11 classes total: the 10 smell classes plus Non-smell class). This configuration reflects real-world deployment scenarios where the model must distinguish between clean code and various smell types.

Table II presents comprehensive performance comparison across both datasets. The results reveal substantial performance improvements when including non-smelly data:

Performance Comparison:

Smelly-Only (DatasetA): Best accuracy 71% (multiple models including standalone GCN), best F1-score 64% (GCN + RNN method II), performance plateau with most models achieving 54-71% accuracy, high variance with F1-scores ranging from 8% to 64%.

Combined Data (DatasetB): Best accuracy 95% (GCN + Transformer method II), best F1-score 89% (GCN + Transformer method II), consistent performance with most method II models achieving 92-95% accuracy, low variance with F1-scores consistently above 82% for top models.

Key Observations:

a) Non-smell Class Provides Crucial Contrast: The inclusion of non-smelly code significantly improves model discrimination capabilities. The performance gap (95% vs 71% accuracy, 24 percentage points) indicates that non-smell samples serve as critical negative examples, helping the model learn distinctive boundaries between clean code and various smell patterns.

b) Reduced Class Overlap: Smelly-only classification suffers from high inter-class confusion, as evidenced by lower F1-scores and high variance across models. When smell classes overlap in feature space without non-smell anchoring, the model struggles to establish clear decision boundaries. The combined approach mitigates this issue by providing a well-separated non-smell class that helps organize the feature space.

c) Method II Benefits More from Combined Data: The performance improvement from DatasetA to DatasetB is more pronounced for Method II (feature concatenation) than Method I (layer integration). Method II models show 20-24 percentage point accuracy gains, while Method I models show 15-20 percentage point gains. This suggests that feature concatenation better leverages the additional discriminative information provided by non-smell samples.

d) Real-world Deployment Implications: The combined dataset configuration better reflects practical deployment scenarios where code smell detection tools must process entire codebases containing both clean and smelly code. The superior performance on DatasetB (95% accuracy, 89% F1-score) demonstrates that our approach generalizes well to realistic use cases.

Based on these results, all subsequent experiments and the final framework adopt the combined data configuration with non-smell samples, ensuring both higher performance and better alignment with real-world deployment requirements.

TABLE II: PERFORMANCE: SMELLY DATA ONLY VS COMBINED DATA

Model	Smelly Only				Combined			
	DatasetA		DatasetB		DatasetA		DatasetB	
	Acc	F1	Acc	F1	Acc	F1	Acc	F1
GCN+BERT-I	0.71	0.43	0.86	0.56	0.41	0.08	0.42	0.08
GCN+Trans-II	0.71	0.44	0.86	0.55	0.95	0.89	0.95	0.89
GCN+RNN-I	0.66	0.24	0.83	0.39	0.91	0.53	0.91	0.53
GCN+CNN-I	0.71	0.41	0.86	0.55	0.89	0.64	0.89	0.67
GCN+BERT-II	0.51	0.09	0.55	0.12	0.95	0.90	0.95	0.83
GCN+GRU-II	0.71	0.41	0.84	0.50	0.94	0.88	0.94	0.88
GCN+LSTM-II	0.70	0.40	0.82	0.47	0.90	0.73	0.90	0.73
GCN+MLP-II	0.68	0.38	0.81	0.47	0.92	0.76	0.92	0.76
GCN+GRU-I	0.67	0.33	0.79	0.48	0.89	0.62	0.89	0.62
GCN+CNN-II	0.66	0.37	0.78	0.41	0.94	0.83	0.94	0.83
GCN+LSTM-I	0.69	0.30	0.83	0.42	0.89	0.56	0.89	0.56
GCN+MLP-I	0.68	0.33	0.79	0.35	0.90	0.65	0.90	0.65
GCN+RNN-II	0.71	0.42	0.83	0.50	0.95	0.86	0.95	0.86
GCN+Trans-I	0.54	0.09	0.56	0.08	0.38	0.05	0.38	0.05

3) Direct Code Smell Classification Results: Standalone GCN showed moderate baseline performance. Several GCN-based integrations using Method II showed comparable or slightly better performance than standalone GCN on Dataset B. GCN combined with Transformer or CNN achieved the same accuracy as GCN, with minor changes in F1-score.

4) Method Comparison (Method II vs Method I): Method II consistently outperformed Method I across all model combinations on Dataset B. Accuracy improvements ranged from 6% to 20%, and F1-score gains were also observed, indicating the advantage of feature concatenation over early fusion approaches.

Conclusion for RQ2 & RQ3: Method II is more effective than Method I for model integration, especially on Dataset B. While not always better than standalone GCN, several combinations demonstrate strong or comparable performance. Our ablation study reveals that explicit feature engineering steps are detrimental to performance, achieving only 67% accuracy compared to 95% for the direct classification approach.

C. RQ4: Comparison with Large Language Models

CodeT5 (Fine-tuned LLM) achieves 97% accuracy and 85% F1-score. GCN + Transformer (Method II) achieves 95% accuracy and 89% F1-score. GCN + LLM Direct Integration

TABLE III: CODE SMELL DETECTION: COMBINED DATA WITH SMOTE

Model Name	DatasetA	F1	DatasetB	F1
GCN + Transformer - II	0.95	0.93	0.95	0.89
GCN + GRU - II	0.94	0.90	0.95	0.88
GCN + BERT - II	0.94	0.89	0.94	0.87
GCN + RNN - II	0.95	0.93	0.94	0.89
GCN + CNN - II	0.94	0.90	0.94	0.88
GCN + MLP - II	0.93	0.88	0.92	0.84
GCN + LSTM - II	0.94	0.90	0.94	0.88
GCN + Transformer - I	0.89	0.85	0.90	0.82
GCN + GRU - I	0.89	0.85	0.90	0.82
GCN + BERT - I	0.90	0.86	0.91	0.83
GCN + RNN - I	0.89	0.85	0.90	0.82
GCN + CNN - I	0.89	0.85	0.90	0.82
GCN + MLP - I	0.89	0.85	0.90	0.82
GCN + LSTM - I	0.89	0.85	0.90	0.82

TABLE IV: FINE-TUNED LLM PERFORMANCE

Model Name	DatasetA	F1	DatasetB	F1
CodeT5	0.97	0.86	0.98	0.85
GraphCodeBERT	0.96	0.84	0.97	0.83
ALBERT	0.96	0.81	0.96	0.80
XLM-RoBERTa	0.96	0.80	0.96	0.79
CodeT5+	0.96	0.86	0.97	0.84
RoBERTa	0.95	0.80	0.95	0.78
XLNet	0.95	0.79	0.95	0.78
CodeBERT	0.95	0.74	0.95	0.72
BERT	0.92	0.64	0.93	0.67

shows 72.79% accuracy and 45.30% F1-score on Dataset A, and 87.11% accuracy and 59.23% F1-score on Dataset B.

D. ROC Curve Comparison

Figure 5 shows the ROC curves comparing standard NLP transformer-based models with Graph Neural Network (GNN) enhanced models.

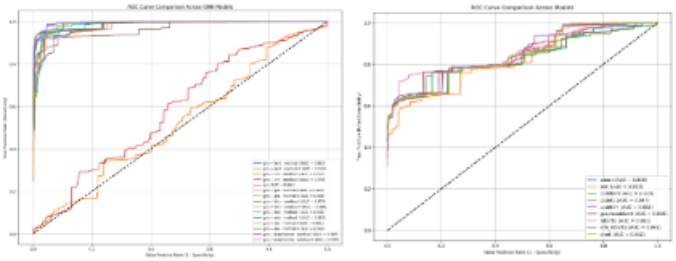

Fig. 5: ROC Curve Comparison Between Standard Transformer Models and GNN-Enhanced Models. AUC scores indicate that GNN-based models significantly outperform traditional models.

Standard transformer-based models achieve AUC values ranging from 0.818 to 0.861, demonstrating moderately strong performance. In contrast, GNN-enhanced models show substantially higher AUC values, reaching up to 0.994, particularly when combining GNN with transformer methods. This dramatic improvement suggests that incorporating graph-based learning structures significantly enhances classification performance by effectively capturing code structural relationships.

Conclusion for RQ4: GCN-Transformer integration achieves competitive performance with specialized LLMs (95% vs 97% accuracy) while providing superior F1-scores (89% vs 85%), indicating better balanced precision-recall characteristics and improved model interpretability.

E. Confusion Matrix Analysis

Figure 6 presents the confusion matrix for the GCN + Transformer (Method II) model on the test set, revealing prediction patterns across all 11 classes.

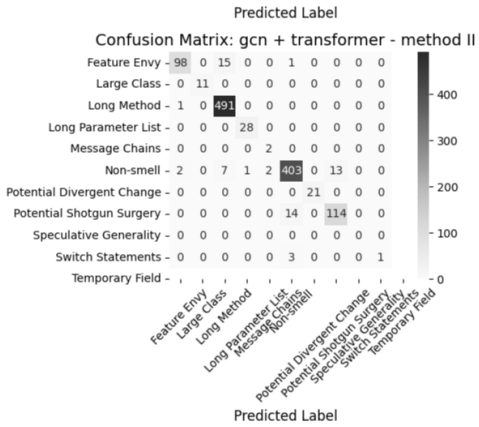

Fig. 6: Confusion matrix for GCN + Transformer (Method II) showing strong diagonal performance with high accuracy across all smell types and Non-smell class.

The matrix demonstrates high accuracy across most classes with particularly strong performance on Non-smell classification, validating our design decision to include non-smelly samples. Minor confusion between structurally similar smell types is expected given overlapping code smell characteristics. The diagonal dominance confirms the model's ability to distinguish between different smell types effectively, supporting the reported 95% accuracy and 89% F1-score.

VI. Discussion

A. Key Findings

1) Per-line graph construction significantly outperforms global vocabulary graphs, providing better generalization.
2) Feature concatenation (Method II) performs better than layer integration (Method I) across all model combinations.
3) Including non-smelly code dramatically improves performance (95% vs 71% accuracy, 24 percentage points), providing critical negative examples for decision boundaries.
4) SMOTE oversampling dramatically improves performance (96.14% vs 86.70% accuracy).

5) GCN-Transformer integration achieves performance comparable to specialized LLMs with better F1-scores and interpretability.
6) End-to-end learning (95% accuracy) significantly outperforms feature engineering (67% accuracy, 28 percentage point gap), demonstrating GCNs effectively learn hierarchical representations without explicit feature extraction.

B. Limitations and Future Work

Limitations include Python-only evaluation, dataset dependency, and computational overhead for large codebases. Future work includes extension to other languages (Java, C++, JavaScript), ensemble methods, optimization for scale, and industrial deployment evaluation.

VII. Conclusion

This study demonstrates that Graph Convolutional Networks integrated with traditional machine learning models provide an effective framework for automated code smell detection. The per-line graph construction approach with feature concatenation integration achieves performance competitive with specialized language models (95% accuracy, 89% F1-score vs CodeT5's 97% accuracy, 85% F1-score) while offering superior interpretability.

Key contributions: (1) Per-line graphs superior to global vocabulary approaches; (2) Including non-smelly samples improves performance by 24 percentage points; (3) Feature concatenation outperforms layer integration; (4) End-to-end learning (95%) outperforms feature engineering (67%); (5) SMOTE improves performance from 87% to 96%; (6) Competitive with state-of-the-art LLMs with better F1-scores.

Practical Implications: The framework offers several practical benefits: (1) **Automation:** Reduces manual code review effort while maintaining high accuracy for CI/CD integration; (2) **Interpretability:** Confusion matrix reveals misclassification patterns, helping prioritize refactoring; (3) **Scalability:** Per-line graphs enable parallel processing for large codebases; (4) **Cost-Effectiveness:** Achieves near-LLM performance without expensive computational resources.

Recommendations for Practitioners: We recommend: Always include non-smelly samples in training datasets; apply SMOTE for class imbalance; prioritize Method II for feature integration; avoid intermediate feature engineering; validate using confusion matrices for targeted refinement.

Threats to Validity: **Internal:** PyExamine may introduce tool-specific biases; BERT embeddings influence representation. **External:** Limited to Python projects from specific domains; generalization to other languages requires investigation. **Construct:** Code smell definitions are context-dependent. **Conclusion:** Statistical significance testing across random seeds would strengthen findings, though substantial performance differences (24-28 percentage points) suggest robust results.

References

[1] E. Hamouda, A. El-Korany and S. Makady, "Smell-ML: A Machine Learning Framework for Detecting Rarely Studied Code Smells," *IEEE Access*, vol. 13, pp. 12966-12980, 2025.

[2] S. Karthikeyan and T. Kathirvalavakumar, "Genetic Algorithm Based Over-Sampling with DNN in Classifying the Imbalanced Data Distribution Problem," *Indian Journal of Science and Technology*, vol. 16, no. 8, pp. 547-556, 2023.

[3] Z. Lu, P. Du and J.-Y. Nie, "VGCN-BERT: Augmenting BERT with Graph Embedding for Text Classification," in *Proceedings of the 42nd European Conference on Information Retrieval (ECIR)*, Springer International Publishing, pp. 369-382, 2020.

[4] K. Shivashankar and A. Martini, "PyExamine: A Comprehensive, Un-Opinionated Smell Detection Tool for Python," arXiv preprint arXiv:2501.18327, 2025.

[5] M. Zhang, J. Jia, L. F. Capretz, X. Hou and H. Tan, "Graph neural network-based long method and blob code smell detection," *Science of Computer Programming*, vol. 243, pp. 103284, 2025.

[6] Y. Zhang, C. Ge, S. Hong, R. Tian, C. Dong and J. Liu, "DeleSmell: Code smell detection based on deep learning and latent semantic analysis," *Knowledge-Based Systems*, vol. 255, pp. 109737, 2022.

[7] Y. Zhang, C. Ge, H. Liu and K. Zheng, "Code smell detection based on supervised learning models: A survey," *Neurocomputing*, vol. 565, pp. 127014, 2024.

[8] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*, 2nd ed., Addison-Wesley Professional, 2018.

[9] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk, "Detecting bad smells in source code using change history information," in *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, pp. 268-278, 2013.

[10] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and P. S. Yu, "A comprehensive survey on graph neural networks," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 32, no. 1, pp. 4-24, 2021.

[11] B. Al-Shaaby, A. S. Alshehri, and L. Benhlima, "Python code smells detection using conventional machine learning models," *PeerJ Computer Science*, vol. 9, pp. e1370, 2023.

[12] M. Moha, Y.-G. Guéhéneuc, L. Duchien, and A.-F. Le Meur, "DECOR: A method for the specification and detection of code and design smells," *IEEE Transactions on Software Engineering*, vol. 36, no. 1, pp. 20-36, 2010.

[13] S. M. H. Rizvi, R. Imran, and A. Mahmood, "Text Classification using Graph Convolutional Networks: A Comprehensive Survey," *ACM Computing Surveys*, 2024.

[14] A. Al-Shaaby, H. Aljamaan, and M. Alshayeb, "Application of Deep Learning for Code Smell Detection: Challenges and Opportunities," *SN Computer Science*, vol. 5, pp. 1-19, 2024.

[15] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," in *Proceedings of the International Conference on Learning Representations (ICLR)*, 2017.

[16] J. Bruna, W. Zaremba, A. Szlam, and Y. LeCun, "Spectral networks and locally connected networks on graphs," in *Proceedings of the International Conference on Learning Representations (ICLR)*, 2014.

[17] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "SMOTE: synthetic minority over-sampling technique," *Journal of Artificial Intelligence Research*, vol. 16, pp. 321-357, 2002.

[18] G. Kovács, "An Investigation of SMOTE Based Methods for Imbalanced Datasets With Data Complexity Analysis," *IEEE Transactions on Knowledge and Data Engineering*, vol. 35, no. 7, pp. 6482-6496, 2023.