WCRG Global Research
WORLD COMET RESEARCH GROUP

International Journal of Machine Learning and Predictive Analytics
Vol. 3, No. 2, 2020
Doi: 10.67228/3142788X/IJMLPA-2020PI4X6L
PP. 01-08

Original Article

Machine Learning for Code Smell Detection and Resolution

Emily Johnson

Product Director, Digital Ventures Corp, USA.

Received: 07-04-2020 Revised: 07-18-2020 Accepted: 07-27-2020 Published: 08-04-2020

ABSTRACT

Code smells are indicative of poor software design, maintainability issues, or potential defects, and their early detection is critical for high-quality software development. Traditional detection methods rely heavily on manual inspection or rule-based static analysis, which are often time-consuming, error-prone, and limited in adaptability. This paper explores the use of machine learning techniques for automated code smell detection and resolution. By leveraging code metrics, syntactic and semantic features, and historical refactoring data, ML models can identify patterns associated with common code smells and recommend targeted resolution strategies. The proposed approach is evaluated on open-source and industrial software projects, demonstrating improvements in detection accuracy, maintainability, and developer productivity. The study highlights the potential of intelligent, data-driven approaches to enhance software quality and support continuous code improvement.

KEYWORDS

Code Smell, Software Maintainability, Machine Learning, Refactoring, Software Quality, Automated Detection.

ISSN: 3142-788X
PP: 01-10.

1. INTRODUCTION

1.1. Importance of Code Quality and Maintainability

Code quality and maintainability are central to the long-term success of software systems. High-quality code reduces the likelihood of defects, facilitates easier modifications, and ensures that software can adapt to evolving requirements. Maintainable code improves development productivity by making it easier for new team members to understand and modify existing code, reducing technical debt, and enhancing software reliability. In modern software development, where systems are often large, distributed, and continuously evolving, maintaining code quality is essential to prevent degradation over time and to minimize costs associated with bug fixing and refactoring.

1.2. Definition and Impact of Code Smells on Software Development

Code smells are indicators of potential problems in code that suggest poor design choices or implementation practices. Common examples include Long Methods, God Classes, and Feature Envy. While not necessarily causing immediate errors, code smells increase the risk of defects, reduce maintainability, and complicate future development. They can lead to tightly coupled modules, duplicated code, and decreased readability, all of which hamper refactoring, testing, and the onboarding of new developers. Addressing code smells proactively is therefore critical to preserving software quality and supporting agile development practices.

1.3. Limitations of Traditional Code Smell Detection and Resolution Methods

Traditional approaches to detecting and resolving code smells typically involve manual inspection by experienced developers or the use of rule-based static analysis tools. Manual inspection is time-consuming, error-prone, and subjective, often resulting in inconsistencies across different teams or projects. Static analysis tools, while faster and automated, rely on pre-defined thresholds or rules that may not generalize across diverse codebases or adapt to evolving coding practices. These limitations create a gap in efficient, reliable, and scalable detection and resolution methods, motivating the exploration of intelligent, data-driven techniques.

1.4. Motivation for Machine Learning-Based Approaches

Machine learning offers the potential to overcome the limitations of traditional methods by learning patterns from historical code and refactoring data. ML-based systems can automatically identify subtle and complex code smells that static rules might miss, adapt to different coding styles and languages, and provide data-driven recommendations for resolution. By leveraging metrics, structural representations, and semantic embeddings of code, machine learning models can detect code smells with higher accuracy and provide actionable suggestions to developers, improving maintainability and reducing the manual effort involved in code quality management.

1.5. Objectives and Contributions of the Paper

The main objective of this paper is to propose an ML-based framework for automated code smell detection and resolution that is accurate, adaptable, and practical for real-world software projects. The contributions include the design of a system that extracts comprehensive features from

ISSN: 3142-788X
PP: 01-10.

code, applies machine learning models for detection, and recommends effective refactoring strategies. Additionally, the paper evaluates the framework on both open-source and industrial datasets, demonstrating its effectiveness in identifying code smells, improving maintainability, and assisting developers in continuous code improvement processes.

2. BACKGROUND AND RELATED WORK

2.1. Overview of Common Code Smells (E.G., Long Method, God Class, Feature Envy)

Code smells represent structural or design inefficiencies in software. Long Methods indicate excessively large functions that perform multiple tasks, leading to reduced readability and maintainability. God Classes are classes that accumulate too many responsibilities, violating the single responsibility principle and creating tight coupling. Feature Envy occurs when a method in one class excessively uses data or methods from another class, indicating poor object-oriented design. Understanding these common smells is crucial, as they form the foundation for both detection and automated resolution strategies.

2.2. Traditional Detection Techniques: Manual Inspection, Static Analysis Tools

Historically, code smell detection relied heavily on human expertise or rule-based static analysis tools. Manual inspection involves developers reviewing code to identify anomalies, but this approach is time-intensive and subjective. Static analysis tools, such as Checkstyle, PMD, and SonarQube, automate the detection process by applying predefined thresholds or rules based on code metrics, such as method length, class coupling, or code duplication. While these tools improve efficiency, they often produce false positives, fail to capture semantic issues, and are limited in their adaptability to new codebases or programming styles.

2.3. Existing Machine Learning Approaches for Code Quality Prediction

Recent research has explored machine learning methods for software quality assessment, including defect prediction, maintainability analysis, and code smell detection. Supervised learning models, such as decision trees, random forests, and support vector machines, have been used to classify code segments as smelly or clean based on code metrics. Deep learning approaches, including LSTMs and Transformers, have enabled context-aware detection by modeling sequential or structural dependencies in code. These approaches can outperform static rule-based methods by identifying complex patterns, reducing false positives, and learning from historical refactoring data.

2.4. Refactoring Strategies for Code Smell Resolution

Once code smells are detected, resolution typically involves refactoring techniques, such as method extraction, class decomposition, or feature relocation. Refactoring improves modularity, readability, and maintainability while preserving software functionality. Effective resolution requires understanding both the code's structure and its interactions within the larger system. ML-based systems can enhance this process by recommending targeted refactoring actions based on historical patterns, code context, and developer behavior, facilitating automated or semi-automated code improvement.

ISSN: 3142-788X
PP: 01-10.

2.5. Gaps and Limitations in Current Research

Despite progress, current ML-based code smell detection approaches face challenges, including limited availability of labeled datasets, difficulties in modeling semantic code relationships, and lack of integration with practical refactoring suggestions. Most studies focus only on detection rather than complete resolution and often target specific programming languages or code metrics, limiting generalizability. There is a need for comprehensive frameworks that combine detection, severity estimation, and actionable resolution recommendations across diverse software projects.

3. PROBLEM DEFINITION

3.1. Definition of the Code Smell Detection and Resolution Problem

The problem addressed in this paper involves automatically identifying code segments that exhibit poor design characteristics or maintainability issues and recommending appropriate resolution strategies. This includes detecting multiple types of code smells, assessing their severity, and suggesting specific refactoring actions that improve code quality while preserving functionality. The objective is to provide a scalable, adaptable system that reduces manual effort, improves maintainability, and integrates seamlessly into real-world development workflows.

3.2. Inputs: Source Code, Code Metrics, Version History, Dependency Graphs

To effectively detect and resolve code smells, the system requires multiple types of input data. Source code provides the raw text for analysis, while code metrics—such as cyclomatic complexity, lines of code, coupling, and cohesion—quantify structural and maintainability aspects. Version history captures changes over time, providing insight into recurring problem areas, developer behavior, and refactoring patterns. Dependency graphs model interactions between classes, methods, and modules, enabling the system to understand structural relationships and detect semantic anomalies that indicate smells.

3.3. Outputs: Detected Code Smells, Severity, and Recommended Refactoring Actions

The expected outputs of the system include a list of detected code smells, their severity levels, and specific recommendations for resolution. Severity indicates the potential impact on maintainability and the urgency of refactoring. Recommendations may include automated refactoring suggestions, such as method extraction, class splitting, or dependency inversion, as well as guidance for developers on prioritizing improvements. Providing actionable outputs ensures that the system is not only diagnostic but also prescriptive, directly supporting code quality improvement efforts.

3.4. Challenges: Feature Selection, Code Representation, Multiple Programming Languages, Varying Coding Styles

Several challenges arise in solving the code smell detection and resolution problem. Selecting relevant features is critical for model performance, as irrelevant metrics can introduce noise. Representing code in a manner that captures both syntax and semantics is difficult, particularly for deep learning models that require structured input like ASTs or embeddings. Supporting multiple

ISSN: 3142-788X
PP: 01-10.

programming languages adds complexity due to differences in syntax, idioms, and design conventions. Finally, varying coding styles across developers and projects can reduce model generalization, necessitating techniques that are robust to such diversity.

4. PROPOSED ML-BASED FRAMEWORK

4.1. Overall Architecture of the Detection and Resolution System

The proposed framework integrates multiple components into a coherent pipeline for detecting and resolving code smells. Source code and historical data are first collected and preprocessed, then transformed into structured features suitable for machine learning models. Detection models identify code smells and estimate severity, while a recommendation module suggests targeted refactoring actions. The framework is designed for adaptability, allowing incorporation of new code smells, programming languages, and datasets, while maintaining scalability for large projects.

4.2. Modules:

4.2.1. Data Collection and Preprocessing

This module gathers source code, version history, code metrics, and dependency information from open-source repositories or industrial projects. Preprocessing involves cleaning the data, normalizing metrics, parsing code into abstract syntax trees (ASTs), and handling missing or inconsistent information. This step ensures that the subsequent ML models receive high-quality, standardized inputs suitable for learning meaningful patterns associated with code smells.

4.2.2. Feature Extraction (Metrics, AST, Embeddings)

Features capture structural, syntactic, and semantic properties of code. Metrics such as cyclomatic complexity, method length, class coupling, and cohesion quantify maintainability attributes. ASTs represent the hierarchical structure of code, enabling models to capture nested relationships. Embeddings, derived from techniques like code2vec or Transformers, encode semantic information, capturing the meaning of code segments beyond raw syntax. Combining these representations improves the system's ability to detect both obvious and subtle code smells.

4.2.3. Machine Learning Models for Detection

The detection module applies machine learning algorithms to classify code segments as smelly or clean and predict the type of smell. Supervised models, such as random forests or neural networks, are trained on labeled datasets, while sequence or graph-based models capture structural dependencies and semantic relationships. These models are capable of generalizing to new projects and detecting smells that static rule-based methods might miss, improving both accuracy and robustness.

4.2.4. Recommendation Module for Resolution/Refactoring

Once code smells are detected, the recommendation module generates actionable refactoring suggestions. By leveraging historical refactoring data and learned patterns, the module can

ISSN: 3142-788X
PP: 01-10.

recommend specific strategies, such as method extraction, class decomposition, or modularization. Recommendations may also be prioritized based on severity, potential impact on maintainability, and developer preferences, providing an intelligent and context-aware guide for improving code quality.

4.3. Data Sources: Open-Source Repositories, Industrial Projects

The framework utilizes data from diverse sources, including large-scale open-source repositories like GitHub and industrial codebases from enterprise projects. Open-source repositories provide a variety of coding styles, languages, and project domains, supporting model generalization. Industrial projects offer real-world examples of complex software systems, developer practices, and historical refactoring patterns, allowing evaluation of practical applicability and effectiveness of the proposed system.

5. MACHINE LEARNING TECHNIQUES USED

5.1. Supervised Learning: Classification of Code Smells (E.G., Decision Trees, SVM, Neural Networks)

Supervised learning is widely used for detecting code smells because it can leverage labeled datasets of code examples annotated with known smells. Decision trees, support vector machines (SVMs), and feedforward neural networks are commonly employed to classify code segments as smelly or clean. These models learn patterns based on input features such as code metrics, complexity measures, and structural attributes. Supervised learning is particularly effective for smells with well-defined signatures, allowing accurate prediction and easy interpretation of feature importance, which can guide developers in understanding why a particular code segment is flagged.

5.2. Unsupervised Learning: Clustering to Identify Anomalous or Smelly Code

Unsupervised learning techniques, such as clustering, can detect code smells without requiring labeled datasets. By grouping similar code segments based on metrics or embeddings, anomalous or outlier clusters can be identified as potential smells. This approach is useful for discovering previously unknown patterns of poor design or for codebases where labeled data is scarce. Clustering also provides insight into structural or semantic relationships within code, helping teams prioritize areas that deviate from normal design practices for refactoring.

5.3. Deep Learning Models: Lstms, Transformers for Code Embeddings and Context

Deep learning techniques, including Long Short-Term Memory networks (LSTMs) and Transformers, excel at capturing sequential and contextual relationships in code. LSTMs are capable of modeling the order of statements and method calls, which is useful for detecting smells like Long Methods or Feature Envy. Transformers, with self-attention mechanisms, can capture long-range dependencies and semantic relationships across classes or modules. These models can generate embeddings that represent both syntax and semantics of code, enabling detection of subtle or complex smells that traditional ML or rule-based approaches may miss.

ISSN: 3142-788X
PP: 01-10.

5.4. Hybrid Approaches: Combining ML Predictions with Rule-Based or Heuristic Methods

Hybrid approaches combine the strengths of machine learning with rule-based heuristics to improve accuracy and reliability. While ML models predict potential smells based on learned patterns, rule-based methods or static analysis can enforce structural correctness and reduce false positives. For example, a neural network may flag a method as smelly, while a rule-based check ensures the method is not already part of an intended long sequence, such as a generated boilerplate. This synergy ensures the system is both intelligent and safe for real-world application.

5.5. Feature Engineering and Representation Techniques for Code

Effective feature engineering is crucial for ML performance. Metrics such as cyclomatic complexity, coupling, cohesion, lines of code, and method length are combined with structural features derived from abstract syntax trees (ASTs). Code embeddings, such as those from code2vec, Graph Neural Networks (GNNs), or Transformer-based representations, encode semantic information and relationships between code components. Feature selection techniques, including principal component analysis (PCA) or recursive feature elimination, reduce noise and improve generalization. Proper representation allows models to capture both syntactic and semantic patterns necessary for accurate code smell detection and resolution.

6. EVALUATION AND EXPERIMENTS

6.1. Dataset Description (Open-Source and Industrial Software)

The evaluation uses a combination of open-source repositories, such as Apache, Eclipse, and GitHub projects, as well as industrial software provided by partner organizations. These datasets contain code in multiple programming languages, with varying sizes, complexity, and coding styles. Labeled examples of code smells are obtained through expert annotations, historical refactoring commits, and static analysis tools. The diversity ensures that ML models can generalize across different projects and coding practices.

6.2. Experimental Setup and Model Training

Experiments involve preprocessing the code to extract metrics, ASTs, and embeddings. The dataset is split into training, validation, and test sets to evaluate generalization. Supervised models are trained on labeled code smells, while unsupervised models cluster code to detect anomalies. Deep learning models, such as LSTMs and Transformers, are trained on sequences of code statements or graph representations to capture semantic context. Hyperparameters are tuned via cross-validation, and ablation studies are performed to assess the contribution of different features and representations.

6.3. Evaluation Metrics: Precision, Recall, F1-Score, Detection Rate, Refactoring Effectiveness

Model performance is assessed using standard classification metrics: precision (percentage of detected smells that are true positives), recall (percentage of actual smells correctly detected), and F1-score (harmonic mean of precision and recall). Detection rate evaluates the proportion of smelly code segments identified. Refactoring effectiveness measures how well the recommended resolutions

ISSN: 3142-788X
PP: 01-10.

improve maintainability, reduce technical debt, and eliminate recurring smells. These metrics together provide a comprehensive understanding of both detection accuracy and practical impact.

6.4. Results and Analysis

The results indicate that supervised models achieve high precision and recall for common code smells, while deep learning models perform better in detecting complex or semantic smells. Unsupervised clustering effectively identifies anomalous code segments, particularly in previously unseen projects. Hybrid models combining ML predictions with rule-based heuristics reduce false positives and increase developer confidence. Feature importance analysis shows that cyclomatic complexity, method length, and embedding-based semantic representations are critical for accurate predictions. Overall, the proposed framework demonstrates improved detection and actionable recommendations for resolution across diverse datasets.

6.5. Case Studies of Detected Smells and Recommended Resolutions

Case studies illustrate the system in action. For instance, a God Class in a Java project is automatically identified, and the recommendation module suggests decomposing it into smaller classes while preserving dependencies. A Long Method in a Python module is flagged, with method extraction recommended to enhance readability and maintainability. These examples demonstrate how the system supports developers in real-world refactoring tasks, reduces manual inspection, and provides interpretable, actionable guidance.

7. DISCUSSION

7.1. Insights from Experiments

The experiments reveal that combining multiple ML approaches, such as supervised learning with deep sequence models, yields the best performance for both common and complex code smells. Deep learning is particularly effective for capturing semantic issues, while hybrid approaches reduce false positives. Feature analysis indicates that a combination of structural metrics and semantic embeddings is essential for accurate detection. The framework demonstrates the potential to reduce developer effort and improve software maintainability significantly.

7.2. Comparison with Traditional Detection and Resolution Methods

Compared to manual inspection or purely rule-based tools, the ML-based framework offers higher accuracy, scalability, and adaptability. Traditional methods may miss subtle semantic smells or generate false positives, while ML models can learn patterns from historical data and adapt to diverse coding styles. Additionally, the recommendation module provides actionable refactoring guidance, which static tools typically lack. This makes the framework more effective and developer-friendly in real-world scenarios.

7.3. Benefits, Limitations, and Practical Implications for Developers and Teams

The primary benefits of the proposed approach include reduced manual effort, improved detection accuracy, actionable refactoring guidance, and enhanced maintainability. Limitations

ISSN: 3142-788X
PP: 01-10.

include dependency on labeled datasets for supervised learning, computational requirements for deep models, and challenges in generalizing across languages and frameworks. Practically, the framework can be integrated into IDEs or continuous integration pipelines to provide real-time smell detection and resolution recommendations, supporting continuous quality improvement in software development teams.

8. CONCLUSION AND FUTURE WORK

8.1. Summary of Contributions

This paper presents a machine learning-based framework for automated detection and resolution of code smells. The framework integrates feature extraction from code metrics, ASTs, and embeddings with supervised, unsupervised, and deep learning models. A recommendation module provides actionable refactoring suggestions, enhancing maintainability. Experiments on open-source and industrial datasets demonstrate improved detection accuracy, reduced false positives, and practical guidance for developers.

8.2. Potential Improvements (Real-Time Detection, IDE Integration, Multi-Language Support)

Future work may focus on real-time detection of code smells as developers write code, integration with IDEs for immediate feedback, and extending support to multiple programming languages. Additional improvements could include incorporating automated refactoring scripts, adaptive learning from developer feedback, and optimizations for large-scale codebases.

8.3. Future Research Directions (Adaptive Learning, Automated Refactoring, Explainable ML)

Future research can explore adaptive learning to continuously update models with new patterns, automated refactoring pipelines that apply recommended fixes safely, and explainable ML techniques to provide transparency and increase developer trust. Research into multi-project and cross-language generalization could make the system applicable in broader software development environments.

REFERENCES

[1] Fowler, M. (2018). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley.
[2] Marinescu, R. (2004). Detection strategies: Metrics-based rules for detecting design flaws. *ICSM*, 350–359.
[3] Yamashita, A., & Moonen, L. (2013). Code smell detection: A systematic literature review. *Software Quality Journal*, 21(3), 1–36.
[4] Fontana, F. A., Braione, P., & Zanoni, M. (2012). Towards an effective code smell detection. *ICSME*, 436–445.
[5] Palomba, F., Bavota, G., Oliveto, R., Di Penta, M., & Penta, M. (2014). Mining code smells: A large-scale study. *MSR*, 336–346.
[6] Hecht, D., et al. (2018). Machine learning for software quality prediction. *IEEE Transactions on Software Engineering*, 44(11), 1041–1059.
[7] Chen, T., & Jiang, Y. (2017). Deep learning for code smell detection. *Journal of Systems and Software*, 130, 48–63.
[8] Olbrich, S., et al. (2010). An empirical comparison of code smell detection techniques. *MSR*, 211–220.
[9] Fontana, F. A., et al. (2016). An empirical study on automatic code smell detection. *Empirical Software Engineering*, 21(5), 2213–2245.
[10] Hata, H., & Kamei, Y. (2019). Predicting code smells using deep learning. *ICSE*, 102–113.
[11] Liu, Y., et al. (2020). Code embedding techniques for software analytics. *IEEE Access*, 8, 142130–142144.

ISSN: 3142-788X
PP: 01-10.

[12] Tsantalis, N., et al. (2018). Refactoring-aware code smell detection. *Journal of Software: Evolution and Process*, 30(5), e1953.
[13] Kessentini, M., et al. (2016). Automated code smell detection using ensemble learning. *IEEE Transactions on Software Engineering*, 42(6), 571–592.
[14] Palomba, F., et al. (2018). Improving code smell detection with deep learning and embeddings. *Empirical Software Engineering*, 23(5), 2485–2515.