Beyond the Code: Unraveling the Applicability of Graph Neural Networks in Smell Detection

Djamel Mesbah[1 2 3], Nour El Madhoun[3 4], Khaldoun Al Agha[2], and Hani Chalouati[1]

[1] Adservio Group, T. Franklin, 100 101 Terr. Boieldieu Ét. 9, 92800, Puteaux, France
{djamel.mesbah,hani.chalouati}@adservio.fr
[2] Université Paris-Saclay, CNRS, Laboratoire Interdisciplinaire des Sciences du Numérique, 91190, Gif-sur-Yvette, France
djamel.mesbah@universite-paris-saclay.fr
alagha@lisn.fr
[3] LISITE Laboratory, ISEP, 10 Rue de Vanves, 92130, Issy-les-Moulineaux, France
{djamel.mesbah, nour.el-madhoun}@isep.fr
[4] Sorbonne Université, CNRS, LIP6, 4 place Jussieu, 75005, Paris, France
nour.el_madhoun@sorbonne-universite.fr

Abstract. Code smells signify suboptimal software design and implementation practices that can severely impact code maintainability. While traditional approaches to code smell detection have largely relied on heuristic and metric-based evaluations, recent advancements have explored the efficacy of Machine Learning (ML) techniques, specifically through the lens of Graph Neural Networks (GNNs) and Abstract Syntax Trees (ASTs). This paper critiques and synthesizes findings from two recent studies that employ these technologies to improve the automated detection of code smells. By tacking a close look to these approaches, we aim to highlight their contributions as well as their limitations within the context of current ML methodologies in software engineering. We provide a comparative analysis of the AST representations and GNN models utilized, exploring how they address the challenges of code smell detection and suggesting directions for future research. Our goal is to check the potential of these models to set new benchmarks in the field.

Keywords: Code Smells · Machine Learning · Graph Neural Networks · Abstract Syntax Trees.

1 Introduction

Code smells refer to bad design and bad practices in the code, indicating signs of code quality problems. These issues particularly impact readability and adaptability, often pointing at the need for refactoring. Detecting and addressing code smells is therefore fundamental to integrating evaluation and improvement into the software evolution process. Software engineering researchers have explored this concept extensively, looking at its roots, implications and detection methods.

Many strategies have been developed to identify code smells in source code. Traditional approaches such as metric-based methods [1] and heuristic-based methods [2] rely heavily on manually defined rules to extract features of relevance from the code. However, these methods face significant challenges as it is difficult to reach consensus on the right rules and metrics and often involves a substantial amount of work. Machine Learning (ML) techniques, including Support Vector Machines, Naive Bayes and Logistic Regression, have also been applied to code smell detection [3–5]. Nevertheless, they still rely on manually crafting specific features and require additional tools and steps for feature extraction and processing. More recently, deep learning models [6] have entered the scene, reducing the need for extensive manual feature engineering as seen in ML techniques.

Despite the emergence of feature-based [7] and token-based [8] techniques for detecting code smells, significant challenges remain in effectively identifying these smells due to the complexity of software structures. Feature-based techniques are strongly supported by manually selected metrics which often lack flexibility and struggle to capture nuanced design flaws. Token-based methods attempt to solve these problems but may lack essential syntactic and semantic relationships within the code. Graph Neural Networks (GNNs) and Abstract Syntax Trees (ASTs) offer a promising solution by taking advantage of the inherent syntactic structure of programs. Indeed, GNNs are being gradually more used in a large spectrum of applications due to their versatility in handling graph like data structure enabling node, edge and graph classification, they are used in recommendation systems [9], molecular biology and chemistry such as molecular property prediction [10], network and security such as intrusion detection [11], malware detection [12] provide a framework that can learn directly from data that can be represented as graphs, which the AST provide through its representation of code snippets after the parsing phase.

Our aim in this paper is to critically analyze two recent studies that use GNNs and ASTs for the detection of code smells [13, 14]. By understanding their methodologies, contributions and challenges, this analysis aims to grasp the effectiveness of the application of ASTs to detect code smells by leveraging the syntactic and semantic structure of source code, as well as using GNNs which allows the model to better understand the complex relationships and data flow in the code.

The paper is organized as follows. Section 1 introduces the study, while Section 2 provides the background on Code Smells, Machine Learning in the software landscape and the Abstract Syntax Tree. Section 3 analyzes the work of the two papers under review. Section 4 presents a detailed discussion of the evaluation results, highlighting the strengths and limitations of the HMML and ASTNN models. Section 5 suggests areas for future investigation. Finally, Section 6 concludes the study.

2 Background

2.1 Code Smells

Code smells, a term coined by Kent Beck [5] in the context of refactoring [15], refer to certain structures in the code that suggest (but do not guarantee) a potential problem. They are often seen as symptoms of poor design or bad programming practices which, without being bugs (as the code is still functioning in their presence), can harm the readability of the code, increase its complexity and hinder its maintainability. Code smells detection is thus important because they can make code more difficult to understand and modify [16], therefore increasing the risk of bugs.

Code smells can be classified into three categories [17–19]:

1. **Implementation Smells**: these arise from poor coding practices and directly affect the quality of individual functions or methods. Examples include long methods that are difficult to understand or maintain, duplicated code that leads to redundancy and high cyclomatic complexity that makes code error-prone and difficult to test.
2. **Design Smells**: these are problems related to design patterns and the modular structure of classes and their interactions. Design smells include cyclic dependencies between classes or modules, long parameter lists that increase complexity and feature envy, when one class makes excessive use of the methods of another.
3. **Architectural Smells**: these are issues that have an impact on the architecture of the system as a whole. They often lead to scalability and maintainability problems. Architectural smells include tightly coupled subsystems that make modifications difficult, inappropriate layering that disrupts modularity and monolithic architectures that make the system inflexible.

2.2 Machine Learning (ML) in Software Engineering

Machine Learning (ML) has revolutionized the field of software engineering, addressing numerous challenges with innovative solutions. In the realm of bug prediction and localization, ML models analyze extensive historical data, code changes, and commit logs to forecast potential defects, thereby reducing debugging efforts and resource allocation significantly [20, 21]. Code smell detection, an essential aspect of maintaining software quality, has similarly benefited from ML techniques. Supervised learning models, including Support Vector Machines, Naive Bayes classifiers, and ensemble methods, have been employed to train on source code metrics, effectively identifying problematic patterns (refer to Section 2.1).

Moreover, ML extends its utility to refactoring recommendations by examining code structures and suggesting enhancements, thus improving maintainability and readability. Natural Language Processing (NLP) techniques in ML have

[5] `https://martinfowler.com/bliki/CodeSmell.htmlr`

found applications in source code summarization and generation, where large language models generate comprehensive code summaries and even complete functional code segments from minimal specifications [22, 23].

In addition to these applications, ML optimizes the software development process itself by analyzing data from version control systems and issue trackers. This analysis helps streamline workflows and predict project timelines, contributing to tasks like code review [24, 25]. GNNs, which exploit the intricate relationships depicted in ASTs, have emerged as particularly innovative tools in this field. By leveraging the hierarchical representation of source code provided by ASTs, GNNs enhance the understanding of syntactic and semantic structures in programs. They have been applied to diverse tasks such as type inference in dynamically typed languages [26], variable naming and misuse detection [27], vulnerability identification [28], and code summarization [29].

2.3 Abstract Syntax Trees (ASTs)

The AST is a hierarchical tree structure generated by the parser during the compilation process. It represents the abstract syntactic structure of the source code, capturing the relationships between different programming constructs such as declarations, assignments and expressions in a language-independent format. Each node in the tree corresponds to a source code construct, such as a function, class or conditional statement and includes key attributes such as variable names or data types. Developers analyze these structures to discover important syntactic features and understand program behavior. An example is shown in Fig. 1 for a function call func(arg1,arg2) which is constructed by creating an EXPR CALL node, so that the left-hand side is the function name and the right-hand side is an unbalanced tree of EXPR ARG nodes.

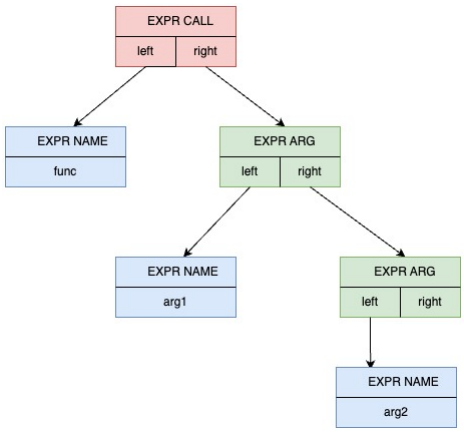

Fig. 1: AST of a Function Call

2.4 Graph Neural Networks

Graph Representation Learning (GRL) is a subcategory of Deep Learning (DL) that handles data structured in a graph. Specifically a graph can be described as: $G = (V, E, \phi_v, \phi_e)$, where a graph G consists of a set of nodes V connected to each other by a set of edges E. Depending on the system and the problem being tackled, the nodes might have features ϕ_v describing the nodes themselves, and the edges may contain features ϕ_e describing the relations between each pair of nodes. The graphs can be cyclic, acyclic, directed, undirected, or even a mixture of these. They can also be homogeneous, where all the nodes are of the same type, or heterogeneous, where we can have a set of types (in recommendation systems it is typical to have a graph consisting of client nodes and product nodes [30]).

The graphs can also be spatio-temporal [31], where there are edges and node features for each timestamp, or dynamic [32], where we have both time-varying structure and time-varying features. Two approaches to structure the graph can be taken:

- Continuous: each addition or removal of a node or edge is an event that is taken into account.
- Discrete: at each timestamp t, we extract a new graph.

The principal idea behind GRL is to extract embeddings from a graph. These embeddings can represent either node properties or structural information of the entire graph. These learned embeddings, rich with semantic information, can then be used in downstream tasks such as node classification [33], link prediction [34], and graph classification [35]. This learning process relies on the Message Passing Neural Network (MPNN) paradigm, which can be decomposed into the steps below:

- Message computation: for each node in the graph, compute the message by aggregating the embeddings of its neighboring nodes along with its own previous embedding. This step involves:
  - Aggregation Function: apply an aggregation function (e.g., sum, mean, max, or a more complex learned function) to combine the embeddings of the neighboring nodes.
  - Message Function: optionally, apply a message function to the aggregated result to transform it further.
- Update Function: apply an update function (e.g., a neural network layer such as a GRU, LSTM, or a simple feed-forward neural network) to the current node's embedding and the computed message to produce the new node representation.

There are two primary settings for learning and inference: inductive and transductive. In the inductive setting, the model is trained on a set of graphs or subgraphs and is capable of generalizing to unseen graphs or nodes that were not present during the training phase. This is particularly useful in applications where the graph structure can change over time, and the model needs to handle

new nodes or edges. The inductive setting emphasizes the model's ability to learn transferable representations that can be applied to different graphs.

In contrast, in the transductive setting, the model is trained and tested on the same graph. The focus is on learning the embeddings for the nodes within this specific graph. The model leverages the entire graph structure during training to make predictions or classifications. This is suitable for scenarios where the graph is static, and the goal is to understand or predict properties within this fixed structure.

3 Analysis of Selected Papers

In this paper, we aim to critically analyze two recent studies that use GNNs and ASTs for code smell detection [13, 14]. The reason for studying these two papers is that the first paper was pioneering in its application of ASTs to detect code smells by leveraging the syntactic and semantic structure of source code to improve detection accuracy. Building on this foundation, the second paper has extended this approach by integrating GNNs and which allows the model to better understand the complex relationships and data flow in the code. These two studies together demonstrate the evolution of AST-based methods where the incorporation of GNNs allegedly results in a more nuanced and efficient detection process that captures code smells across different granularities.

In the first paper [14], the authors address these challenges by introducing a new AST-based method for detecting code smells. Their approach, called ASTNN, starts by generating ASTs from code snippets, then divides each complete AST into several sub-trees, forming sequences of statement trees. These sequences are encoded and a bidirectional GRU [36] with max pooling captures semantic and structural features. This process results in complete vector representations of the code fragments, which are then fed into fully connected layers for final detection. They applied this technique to 500 high-quality Java projects sourced from GitHub and observed that it outperformed existing deep learning models across various granularities of code smells.

In the second paper [13], the authors address existing limitations by introducing a hybrid model with multi-level code representation (HMML). This model starts by parsing the AST from source code, incorporating control and data flow edges to construct the code property graph. A Graph Convolution Network (GCN) [37] is then employed to extract information at both syntactic and semantic levels. Simultaneously, the bidirectional Long Short-Term Memory (LSTM) [38] network with attention analyzes code tokens at the token level. Finally, the predictions from both models are combined using weighted outputs. The entire approach is optimized for multi-label classification, achieving superior results in multi-label code smell detection and specific single-code smell tasks when applied to 100 high-quality Java projects from GitHub.

3.1 Data Collection

Both papers tackle smell detection in the context of Java projects from open source projects on Github. ASTNN aims to detect Insufficient Modularization, Deficient Encapsulation as well as Feature Envy for design smells and Empty Catch Block for implementation smell. In contrast, HMML handles 9 implementation smells. These include Magic Number, Long Identifier, Long Statement, Missing default, Complex Method, Long Parameter List, Complex Conditional, Long Method and Empty catch clause. Indeed, the studies use CodeSplit[6] to split all the projects downloaded from GitHub into class-level and method-level code fragments. These fragments are later processed by Designite [39] to generate smell reports to label all instances and are further parsed through Javalang to get the ASTs out of the code fragments (see Fig. 2).

Furthermore, the two papers reviewed rely on Designite [39] to generate ground-truth labels for code smell detection. Designite is a static code analysis tool designed to detect code smells, design smells and other quality issues in software codebases. It uses a set of predefined heuristics and metrics to analyze source code. In fact, using Designite offers several advantages, including automation and scalability, which are crucial for processing large codebases. This ensures a consistent labeling process, minimizing the risk of human error and variability. However, there are important limitations associated with the exclusive use of Designite: the heuristic nature of Designite can lead to false positives and false negatives, which introduce noise into the training and evaluation datasets. False positives can cause models to learn incorrect patterns, while false negatives can lead to incomplete training data, impairing the model's ability to recognize certain code smells. Additionally, biases or limitations inherent in Designite can be propagated into the machine learning model.

Then, ASTNN [14] decomposes the generated ASTs into statement trees that are stored along with their corresponding labels. While HMML [13] uses the generated ASTs to create graph objects containing the nodes' indices, the edges and the label of the graph (since it is a multi-label task, there are 9 labels, one for each smell). Moreover, the tokenization technique employed to create the vocabularies is not mentioned or detailed in the implementation of the two models. However, it can be inferred that vocabulary creation relied on a standard corpus. ASTNN has a single vocabulary, while HMML has two: one for the token-based approach (strictly semantic) and the other for the graph-based approach (strictly syntactic).

3.2 Evaluation and Experimental Results

The evaluation was made between ASTNN [14], HMML [13] (see Fig. 3 for an overview) and Random Forest [40], with Random Forest chosen as a baseline because tree methods were previously the state-of-the-art models for smell detection [41]. The primary metrics used to assess the performance of the models

[6] `https://github.com/tushartushar/CodeSplitJava`

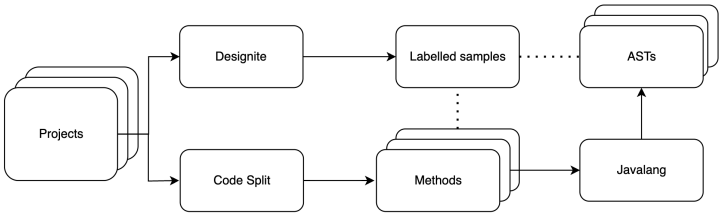

Fig. 2: Data Preprocessing Phases

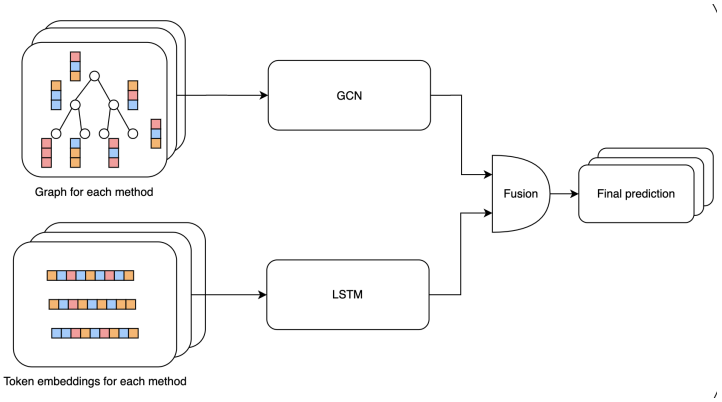

Fig. 3: Model Architecture Overview

are precision, recall and F1 measure, which provide an overview of the accuracy and completeness of the code smell detection. To statistically validate the differences in performance between the models, the Wilcoxon signed-rank test [42] was employed. This non-parametric test is used to compare two related samples to assess if their median ranks differ significantly.

The evaluation of the three models—HMML, Random-Forest and ASTNN across various code smells demonstrates notable differences in their performance, as shown in Table 1. HMML generally outperforms the other models in terms of F1 scores for most code smells, indicating a balanced trade-off between precision and recall. More specifically, HMML model performs exceptionally well in detecting the "Magic Number"(P: 0.97, R: 0.93, F1: 0.95), the "Missing Default" (P: 0.98, R: 0.99, F1: 0.99) and the "Complex Method" (P: 0.82, R: 0.66, F1: 0.73).

In contrast, the Random-Forest model shows varying performance, often characterized by relatively high precision but lower recall leading to lower F1 scores compared to HMML. For example, Random-Forest performs decently in detecting "Magic Number" (P: 0.89, R: 0.35, F1: 0.50) but it struggles significantly with smells such as "Complex Method" and "Long Method." The high precision but low recall suggests that while Random-Forest can accurately iden-

tify certain code smells, it is missing a substantial number of instances resulting in many false negatives. ASTNN generally performs better than Random-Forest but not as well as HMML in terms of F1 scores. It has strong recall but often lower precision, resulting in lower F1 scores for smells such as "Complex Conditional" (P: 0.94, R: 0.21, F1: 0.34) and "Empty Catch Clause" (P: 0.61, R: 0.11, F1: 0.18). This trend indicates that ASTNN is capable of detecting many true positives, but also generates a significant number of false positives.

Indeed, a closer examination of specific code smells reveals further details. For the "Magic Number", HMML achieves the highest F1 score (0.95), indicating excellent performance. This can be attributed to its ability to effectively capture the syntactic and semantic nuances in the code, which are key to accurately identifying this type of smell. ASTNN also performs relatively well in this category (F1: 0.62) while Random Forest lags behind (F1: 0.50) despite its high precision. This discrepancy underlines the importance of balanced recall in addition to precision. Moreover, for the "Long Identifier", ASTNN excels with the highest precision (0.85) and F1 score (0.44), suggesting that its token-based approach is particularly effective in identifying long identifiers. Howecer, HMML and Random-Forest have lower F1 scores (0.49 and 0.43 respectively), indicating that it is difficult to balance precision and recall for this specific smell.

In the "Complex Method" evaluation, HMML came out on top with an F1 score of 0.73, while Random-Forest and ASTNN show significantly lower performance (0.26 and 0.34 respectively). HMML's superior performance in this category may be due to its hybrid approach which combines graph convolution networks and bidirectional long-short term memory networks to capture both structural and token-level information. In fact, in overall multiple smell detection, HMML outperformed both Random-Forest and ASTNN with an F1 score of 0.78, underlining its robustness in handling multiple code smells simultaneously. ASTNN shows moderate performance with an F1 score of 0.62 while Random-Forest is the lowest performer (F1: 0.45). HMML's consistently high performance across different smells can be attributed to its ability to integrate multi-level code representations, enabling it to capture a wide range of code features.

In conclusion, HMML demonstrates superior performance for most code smells, indicating its effectiveness in terms of both precision and recall. The model's balanced approach to syntactic and semantic analysis enables it to excel at detecting complex smells and handling multi-label detection tasks. Random-Forest, while having high precision in some cases, struggles with recall, resulting in poorer overall performance. ASTNN, although better than Random-Forest, is still outperformed by HMML, mainly due to its low precision despite strong recall. These results suggest that HMML is the most reliable of the three models for detecting code smells in Java projects, with performance balanced between the different types of smells.

Code smells	HMML			Random-Forest			ASTNN		
	P	R	F1	P	R	F1	P	R	F1
Magic Number	0.97	0.93	0.95	0.89	0.35	0.50	0.67	0.57	0.62
Long Identifier	0.44	0.55	0.49	0.52	0.36	0.43	0.85	0.30	0.44
Long Statement	0.73	0.60	0.66	0.90	0.35	0.50	0.84	0.68	0.75
Missing default	0.98	0.99	0.99	0.96	0.29	0.44	0.71	0.23	0.34
Complex Method	0.82	0.66	0.73	0.92	0.15	0.26	0.71	0.23	0.34
Long Parameter List	0.81	0.60	0.69	1.00	0.29	0.46	0.86	0.60	0.71
Complex Conditional	0.68	0.58	0.63	0.96	0.14	0.24	0.94	0.21	0.34
Long Method	0.67	0.41	0.51	1.00	0.09	0.16	0.83	0.69	0.76
Empty catch clause	0.51	0.30	0.38	0.86	0.08	0.15	0.61	0.11	0.18
Multi smells	0.83	0.74	0.78	0.90	0.30	0.45	0.76	0.52	0.62

Table 1: Comparison of precision (P), recall (R), and F1 scores for HMML, Random-Forest, and ASTNN across different code smells.

4 Discussion

The evaluation results indicate that while HMML generally outperforms Random-Forest and ASTNN for most code smells, the dataset used has notable limitations and biases. Specifically the use of Designite for ground-truth labeling introduces potential biases due to its heuristic-based nature, which may lead to false positives and false negatives. These inaccuracies can propagate through the models, affecting their ability to generalize and accurately detect code smells in various codebases. An alternative approach, such as manual labeling or the use of multiple static analysis tools, could potentially provide more accurate and reliable ground truth data, reducing inherent bias and improving model performance.

Despite HMML's overall superior performance, both HMML and ASTNN show significant shortcomings in the detection of certain code smells. For example, both models struggle to detect "Long Identifier" and "Complex Method" smells, suggesting that these models may not effectively capture the nuanced patterns associated with these specific smells. The relatively lower performance on these smells highlights the challenges of creating comprehensive feature representations that can generalize across different types of code smells. In addition, the token-based nature of ASTNN, while effective in capturing semantic information, can miss important structural details, leading to lower precision in some cases. On the other hand, HMML's hybrid approach, which combines syntactic and semantic analysis, offers a more balanced performance but remains insufficient in some areas due to the inherent complexity of smells and potential noise in the training data.

Moreover, the dataset, consisting of 100 high-quality Java projects from GitHub, may not fully represent the diversity of real-world software projects. This limitation could affect the generalizability of the models to other program-

ming languages or less well-maintained projects, underlining the need for a more diverse and representative dataset for future research.

In summary, while HMML demonstrates strong overall performance, the results also reveal significant limitations in both the dataset and the models. Addressing these limitations through improved ground-truth labeling methods, more diverse datasets and improved feature representations could lead to more effective and generalizable code smell detection models.

5 Future Work

For the future, several promising avenues can be pursued to improve code smell detection. One interesting approach is to tackle the problem in a self-supervised way. In this approach, a dataset without explicitly labeled smells can be used to train models to integrate the code into a latent space. Once this integration is achieved, outlier detection methods can be applied to identify code smells, taking advantage of the fact that these smells are generally rare anomalies in the dataset. This could reduce reliance on heuristic tools such as Designite and provide a more robust and unbiased means of detecting code smells.

Furthermore, the integration of advanced embedding techniques like code2vec could significantly improve the semantic representation of code features. By embedding code snippets into meaningful vector representations, code2vec [43] can capture intricate semantic relationships within the code, improving the model's ability to detect smells at a deeper level.

Another potential avenue for future research is to extend the analysis beyond ASTs to include Control Flow Graphs (CFGs) [44] [45]. CFGs offer a detailed view of execution paths within a program, providing valuable information on the dynamic behavior of code. By combining AST-based syntax analysis with CFG-based control flow analysis, models could achieve a more complete understanding of the code, which could lead to more accurate smell detection.

6 Conclusion

In this paper, we analyzed two advanced methods for detecting code smells, HMML and ASTNN. We found that the HMML model, which combines syntactic and semantic analysis, generally outperforms the other model for most code smells. However, both HMML and ASTNN have limitations, particularly in the detection of certain smells such as "Long Identifier" and "Complex Method". Future research should focus on improving feature representation and model training, exploring self-supervised learning, advanced integration techniques such as code2vec and incorporating CFGs to improve detection capabilities and overcome current limitations.

References

1. Salehie, M., Li, S., Tahvildari, L.: A metric-based heuristic framework to detect object-oriented design flaws. In: 14th IEEE International Conference on Program Comprehension (ICPC'06). pp. 159–168. IEEE (2006)
2. Moha, N., Guéhéneuc, Y.G., Duchien, L., Le Meur, A.F.: Decor: A method for the specification and detection of code and design smells. IEEE Transactions on Software Engineering **36**(1), 20–36 (2009)
3. Panigrahi, R., Kumar, L., et al.: Application of naïve bayes classifiers for refactoring prediction at the method level. In: 2020 International Conference on Computer Science, Engineering and Applications (ICCSEA). pp. 1–6. IEEE (2020)
4. Kaur, A., Jain, S., Goel, S.: A support vector machine based approach for code smell detection. In: 2017 International Conference on Machine Learning and Data Science (MLDS). pp. 9–14. IEEE (2017)
5. Jain, S., Saha, A.: Improving performance with hybrid feature selection and ensemble machine learning techniques for code smell detection. Science of Computer Programming **212**, 102713 (2021)
6. Liu, H., Jin, J., Xu, Z., Zou, Y., Bu, Y., Zhang, L.: Deep learning based code smell detection. IEEE transactions on Software Engineering **47**(9), 1811–1837 (2019)
7. Alazba, A., Aljamaan, H.: Code smell detection using feature selection and stacking ensemble: An empirical investigation. Information and Software Technology **138**, 106648 (2021)
8. AbuHassan, A., Alshayeb, M., Ghouti, L.: Software smell detection techniques: A systematic literature review. Journal of Software: Evolution and Process **33**(3), e2320 (2021)
9. Wu, S., Sun, F., Zhang, W., Xie, X., Cui, B.: Graph neural networks in recommender systems: a survey. ACM Computing Surveys **55**(5), 1–37 (2022)
10. Wieder, O., Kohlbacher, S., Kuenemann, M., Garon, A., Ducrot, P., Seidel, T., Langer, T.: A compact review of molecular property prediction with graph neural networks. Drug Discovery Today: Technologies **37**, 1–12 (2020)
11. Bilot, T., El Madhoun, N., Al Agha, K., Zouaoui, A.: Graph neural networks for intrusion detection: A survey. IEEE Access (2023)
12. Bilot, T., Madhoun, N.E., Agha, K.A., Zouaoui, A.: A survey on malware detection with graph representation learning. arXiv preprint arXiv:2303.16004 (2023)
13. Li, Y., Zhang, X.: Multi-label code smell detection with hybrid model based on deep learning. In: SEKE. pp. 42–47 (2022)
14. Xu, W., Zhang, X.: Multi-granularity code smell detection using deep learning method based on abstract syntax tree. In: SEKE. pp. 503–509 (2021)
15. Du Bois, B., Demeyer, S., Verelst, J.: Refactoring-improving coupling and cohesion of existing code. In: 11th working conference on reverse engineering. pp. 144–151. IEEE (2004)
16. Santos, J.A.M., Rocha-Junior, J.B., Prates, L.C.L., Do Nascimento, R.S., Freitas, M.F., De Mendonça, M.G.: A systematic review on the code smell effect. Journal of Systems and Software **144**, 450–477 (2018)
17. Sharma, T., Singh, P., Spinellis, D.: An empirical investigation on the relationship between design and architecture smells. Empirical Software Engineering **25**, 4020–4068 (2020)
18. Mumtaz, H., Singh, P., Blincoe, K.: A systematic mapping study on architectural smells detection. Journal of Systems and Software **173**, 110885 (2021)

19. Sharma, T., Efstathiou, V., Louridas, P., Spinellis, D.: Code smell detection by deep direct-learning and transfer-learning. Journal of Systems and Software **176**, 110936 (2021)
20. Pandey, S.K., Mishra, R.B., Tripathi, A.K.: Bpdet: An effective software bug prediction model using deep representation and ensemble learning techniques. Expert Systems with Applications **144**, 113085 (2020)
21. Lam, A.N., Nguyen, A.T., Nguyen, H.A., Nguyen, T.N.: Bug localization with combination of deep learning and information retrieval. In: 2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC). pp. 218–229. IEEE (2017)
22. Wang, Y., Le, H., Gotmare, A.D., Bui, N.D., Li, J., Hoi, S.C.: Codet5+: Open code large language models for code understanding and generation. arXiv preprint arXiv:2305.07922 (2023)
23. Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H.P.d.O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., et al.: Evaluating large language models trained on code. arXiv preprint arXiv:2107.03374 (2021)
24. Tufano, R., Masiero, S., Mastropaolo, A., Pascarella, L., Poshyvanyk, D., Bavota, G.: Using pre-trained models to boost code review automation. In: Proceedings of the 44th international conference on software engineering. pp. 2291–2302 (2022)
25. Li, H.Y., Shi, S.T., Thung, F., Huo, X., Xu, B., Li, M., Lo, D.: Deepreview: automatic code review using deep multi-instance learning. In: Advances in Knowledge Discovery and Data Mining: 23rd Pacific-Asia Conference, PAKDD 2019, Macau, China, April 14-17, 2019, Proceedings, Part II 23. pp. 318–330. Springer (2019)
26. Allamanis, M., Barr, E.T., Ducousso, S., Gao, Z.: Typilus: Neural type hints. In: Proceedings of the 41st acm sigplan conference on programming language design and implementation. pp. 91–105 (2020)
27. Allamanis, M., Brockschmidt, M., Khademi, M.: Learning to represent programs with graphs. arXiv preprint arXiv:1711.00740 (2017)
28. Zhou, Y., Liu, S., Siow, J., Du, X., Liu, Y.: Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. Advances in neural information processing systems **32** (2019)
29. LeClair, A., Haque, S., Wu, L., McMillan, C.: Improved code summarization via a graph neural network. In: Proceedings of the 28th international conference on program comprehension. pp. 184–195 (2020)
30. Fan, S., Zhu, J., Han, X., Shi, C., Hu, L., Ma, B., Li, Y.: Metapath-guided heterogeneous graph neural network for intent recommendation. In: Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining. pp. 2478–2486 (2019)
31. Jin, G., Liang, Y., Fang, Y., Shao, Z., Huang, J., Zhang, J., Zheng, Y.: Spatio-temporal graph neural networks for predictive learning in urban computing: A survey. IEEE Transactions on Knowledge and Data Engineering (2023)
32. Zhang, M., Wu, S., Yu, X., Liu, Q., Wang, L.: Dynamic graph neural networks for sequential recommendation. IEEE Transactions on Knowledge and Data Engineering **35**(5), 4741–4753 (2022)
33. Xiao, S., Wang, S., Dai, Y., Guo, W.: Graph neural networks in node classification: survey and evaluation. Machine Vision and Applications **33**(1), 4 (2022)
34. Li, J., Shomer, H., Mao, H., Zeng, S., Ma, Y., Shah, N., Tang, J., Yin, D.: Evaluating graph neural networks for link prediction: Current pitfalls and new benchmarking. Advances in Neural Information Processing Systems **36** (2024)
35. Errica, F., Podda, M., Bacciu, D., Micheli, A.: A fair comparison of graph neural networks for graph classification. arXiv preprint arXiv:1912.09893 (2019)

36. Cho, K., Van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., Bengio, Y.: Learning phrase representations using rnn encoder-decoder for statistical machine translation. arXiv preprint arXiv:1406.1078 (2014)
37. Kipf, T.N., Welling, M.: Semi-supervised classification with graph convolutional networks. arXiv preprint arXiv:1609.02907 (2016)
38. Hochreiter, S., Schmidhuber, J.: Long short-term memory. Neural computation **9**(8), 1735–1780 (1997)
39. Sharma, T., Mishra, P., Tiwari, R.: Designite: A software design quality assessment tool. In: Proceedings of the 1st International Workshop on Bringing Architectural Design Thinking into Developers' Daily Activities. pp. 1–4 (2016)
40. Guggulothu, T., Moiz, S.A.: Code smell detection using multi-label classification approach. Software Quality Journal **28**(3), 1063–1086 (2020)
41. Arcelli Fontana, F., Mäntylä, M.V., Zanoni, M., Marino, A.: Comparing and experimenting machine learning techniques for code smell detection. Empirical Software Engineering **21**, 1143–1191 (2016)
42. Woolson, R.F.: Wilcoxon signed-rank test. Encyclopedia of Biostatistics **8** (2005)
43. Alon, U., Zilberstein, M., Levy, O., Yahav, E.: code2vec: Learning distributed representations of code. Proceedings of the ACM on Programming Languages **3**(POPL), 1–29 (2019)
44. Ma, W., Liu, S., Lin, Z., Wang, W., Hu, Q., Liu, Y., Zhang, C., Nie, L., Li, L., Liu, Y.: Lms: Understanding code syntax and semantics for code analysis (2023), `https://api.semanticscholar.org/CorpusID:267522763`
45. Xian, Z., Huang, R., Towey, D., Fang, C., Chen, Z.: Transformcode: A contrastive learning framework for code embedding via subtree transformation. IEEE Transactions on Software Engineering (2024)