2023 Innovations in Intelligent Systems and Applications Conference (ASYU) | 979-8-3503-0659-0/23/$31.00 ©2023 IEEE | DOI: 10.1109/ASYU58738.2023.10296577

# Detecting Code Smell with a Deep Learning System

Ali Nizam
*Department Of Software Engineering,*
*Fatih Sultan Mehmet Vakif University*
İstanbul, Turkey
ali.nizam@fsm.edu.tr

Muhammed Yahya Avar
*Department Of Computer Engineering,*
*Fatih Sultan Mehmet Vakif University*
İstanbul, Turkey
muhammed.avar@stu.fsm.edu.tr

Ömer Kaan Adaş
*Department Of Computer Engineering,*
*Fatih Sultan Mehmet Vakif University*
İstanbul, Turkey
omerkaan.adas@stu.fsm.edu.tr

Ahmet Yanık
*Department Of Computer Engineering,*
*Fatih Sultan Mehmet Vakif University*
İstanbul, Turkey
ahmet.yanik@stu.fsm.edu.tr

***Abstract*— Code smell detection is one of the most significant issues in the software industry. Metric-based static code analysis tools are used to detect undesirable coding practices known as code smells and guide refactoring requirements. Furthermore, the usage of deep learning-based techniques has emerged in code analysis tasks. The line and block level detection capability of metric-based tools provides an advantage over deep learning system systems. This study aims to develop a deep learning-based system for inter-procedural code smell detection supporting line and block of code. We created an experimental dataset by gathering code from GitHub repositories and detecting code smell on these codes using the metric-based SonarQube tool. Recurrent neural networks and transformers implementations of deep neural networks were applied to detect code smells. We also employed cosine similarity and k-Nearest Neighbor machine learning techniques for a comprehensive comparison. The proposed system achieves an average accuracy of approximately 80%. These findings indicate that the proposed system can help software teams in identifying potential interprocedural code smells.**

***Keywords—Code smell, machine learning, deep learning***

## I. Introduction

Enhancing code quality is based on code smell detection and refactoring which are important methods for providing and controlling quality attributes [1]. Code smells reflect improper design qualities in object-oriented systems, such as method cohesion and complexity, correct placement of methods in classes, and polymorphic problems, such as long method, feature envy, and state checking [2]. Static analysis tools are widely used to identify quality issues by employing source code metrics and thresholds to detect faults [3] or predict defects [4].

Machine learning-based models can detect software quality problems [5]. However, they require huge preprocessing and feature extraction efforts that create obstacles to practical adaptation [6]. They disregard the context that may imply important quality issues and are negatively affected by missing values [7].

Deep neural network (DNN) based code analysis techniques demonstrate high performance in some areas of code analysis [5]. NLP-based embedding methods have been developed for the low-dimensional vector representations of code, called code embedding. They support analyzing code with machine learning and DNN techniques [8]. Code embedding is used for method naming [9], bug localization [10], and legacy code migration [11]. However, existing approaches have the limitations of intraprocedural embedding [8], [9], [11]. Thus, DNN systems have a significant disadvantage compared to metric-based systems that can perform line and block-based code analysis.

This study aims to identify code smells at the line and block level by designing and implementing machine-learning and DNN approaches. The proposed system supports the detection of code smells to improve code quality. The inputs to our model are a code snippet and a corresponding code smell tag obtained from code analysis of open-source GitHub repositories with SonarQube [12]. We have developed a block and line-based vector representation using the abstract syntax tree (AST) to remove the limitations of intraprocedural embedding using the Bidirectional Encoder Representations from Transformers (BERT) model. We evaluated the effectiveness of our approach using repositories on GitHub as an experimental dataset. We employed cosine similarity, K-nearest neighbor (kNN), and DNN-based techniques and assess their accuracy and F1 score.

Our study provides the *following contributions*: (1) A new model to detect line and block level code smells. (2) Testing the effectiveness of different parameters when converting to AST (3) Comparing the results of machine learning and DNN techniques. In addition, the effects of AST components to be included in the code analysis process were also examined.

The remainder of the paper is organized as follows. Section 2 presents related work on smell detection literature. Section 3 introduces the method and details of the experimental study from creating a dataset to the machine learning and DNN techniques. Section 4 presents an analysis and discussion of the results and answers to the research question. Finally, Section 5 presents conclusions and future work.

## II. Background and Related Work

In this section, we present the background and current state of code analysis by comparing several existing approaches. In addition, we investigate applications of DNN to understand how they are applied to code smell detection.

### A. Code Smell Detection Methods

*Metric-based methods* employ the computation of source code metrics and apply a threshold to detect faults [3] or predict defects [4]. A combination of software metrics constructs metrics-based rules to quantify defects; however, identifying the best threshold values is difficult for them [13].

*Machine learning* utilizes the code smell detection problem that requires data selection, the formalization of the input and output, and algorithms to analyze the statistical properties of the code context [14]. Many techniques have been employed in this area including neural networks [7], Bayesian belief networks [15], and binary logistic regression [16]. The machine learning process involves the following steps: building a code dataset including manual labeling, extracting metrics, determining data and algorithm characteristics, and training and testing [17]. Machine learning techniques require preprocessing to extract features of code to detect code smells (Sharma et al., 2021).

*Deep learning* automatically creates abstract features without specific metric-set specifications [6]. Convolutional Neural Network and RNN methods are used to create code embedding [9], source code generation [18], software vulnerability detection [19], and smell detection [6], [20].

### B. Source Code Representation Models

Code embedding represents code semantics with vector representations for analysis [8]. Embedding support a variety of program analysis tasks, such as code summarization and semantic labeling [8]. There are many code representation models, such as Code2Vec [9], PathPair2Vec [5], and Code2seq [2]. Code2Vec creates a single fixed-length code vector from a code snippet to predict the semantic properties of code [9].

### C. Deep Learning in Code Analysis

DL has multiple processing layers and each layer learns data representations with complicated mappings [21]. DNN models on code evaluation provide important advantages.

DNN techniques require less human labor and show better accuracy than traditional methods because of their abstract high-learning capabilities [10]. DNN-based automatic selection of textual and other features significantly improves accuracy and outperforms well-known methods for detecting code smells [20]. Thus, DNN can detect code smells using only code context without specific feature sets specifications, such as metrics and rules [6].

Recurrent Neural Network (RNN) models are frequently used in this area because their structures support understanding the sequential structure of the code. RNN stores previous input values in the form of activations of feedback connections [22] RNN can store only relevant information and forget others to make predictions by evaluating the importance of information.

## III. Methods

First, we collected data from open GitHub Repositories and detected code smells with the metric-based SonarQube tool to build a code smell dataset. Then this whole code was converted to AST and smelly code parts were separated and converted to the embedding. Cosine similarity and kNN machine learning techniques and Transformers, and LSTM-based deep learning networks were used for data analysis as shown in Fig. 1.

The AST structure consists of many components such as expressions, tokens, uncompiled parts (trivia), and variable names. The effect of taking different components as input on the result was examined comparatively. Especially the effect of the existence of variable names on the results is examined.

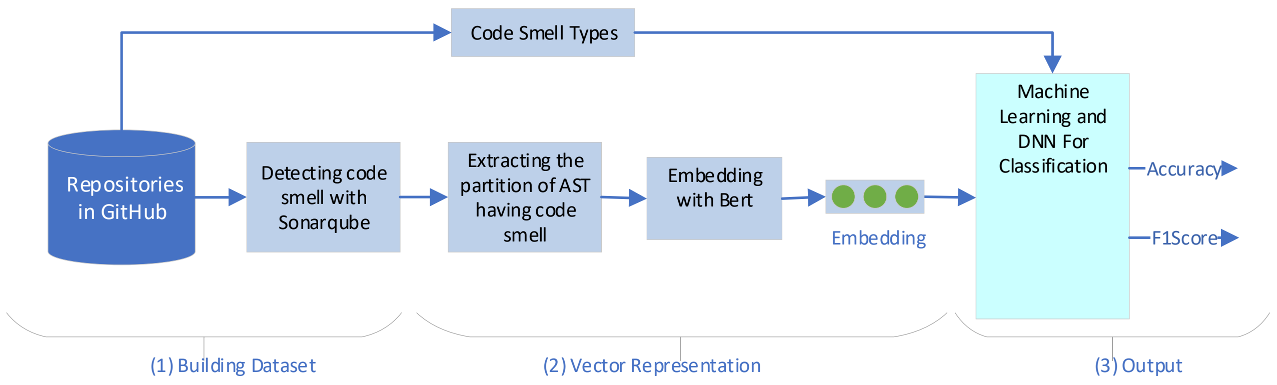

Code Smell Types
Repositories in GitHub
Detecting code smell with Sonarqube
Extracting the partition of AST having code smell
Embedding with Bert
Embedding
Machine Learning and DNN For Classification
Accuracy
F1Score
(1) Building Dataset
(2) Vector Representation
(3) Output

Fig. 1. System model

## A. Data Collection

We have prepared an experimental dataset including AST-code smell type label pairs. Synthetic auto-generated labels have been used for code analysis [9], [23], [24] [25], [26], [27] in the literature. However, synthetic code datasets suffer from duplication issues [28] and biases [29] that degrade generalization. Thus, we preferred to collect code data from GitHub repositories.

SonarQube Community Edition version was used for code smell detection, considering the output formats, code fragments examined, software language compatibility, and code smells. Small applications have been developed in the middleware to automate the steps of getting data from this tool. Since the code smell type detected by SonarQube contains too many details, they are collected under groups as listed in Table I.

TABLE I. SMELL TYPE AND DEFINITIONS

| Smell Type | Definition |
|---|---|
| Refactor | A code segment that may have an impact on the overall flow of the program and requires large changes and careful examination |
| Adjustment | A code segment that requires minor changes or review |
| Remove | A code segment that is unused or would be more suitable for the overall program if deleted |
| Rename | Code segments that do not comply with naming conventions or contain identifiers that would benefit from being renamed. |
| Deficient | Code segments that have needs such as addition, specification, completion, or contain deficiencies. |

## B. Creating Code Embeddings

Pre-processing was applied to the received AST data. Line numbers in the ASTs have been removed so that they do not affect the quality of the algorithms. Then the bad lines of code are encoded, and vocabulary is generated. The input definition includes a $<Embedding, SmellType>$ pairs.

BERT is a language model that captures contextual information by considering the surrounding words in a sentence [30]. BERT utilizes a transformer-based architecture to pre-train large amounts of unlabeled text data, learning general language representations. During pre-training, it uses a masked language model objective to predict masked words in sentences, enabling it to understand the contextual relationships between words.

## C. Machine Learning Techniques

Cosine similarity used a measure used to determine the similarity between two vectors in a multi-dimensional space. It calculates the cosine of the angle between the vectors, which represents similarity.

kNN is an algorithm used in machine learning and is used for classification and regression tasks in pattern recognition problems. The kNN algorithm is based on proximity calculations to predict new data points by considering the labels or values of sample data points.

## D. Deep Learning Techniques

Long Short-Term Memory (LSTM)[22] is an artificial neural network architecture used in the field of deep learning. LSTM, a type of RNN, is particularly effective in processing data sequences with long-term dependencies, such as time series or text data. The main purpose of LSTM is to solve the "long-term dependency problem" encountered by RNNs.

Transformers [31] is a deep learning model that has achieved great success in fields such as natural language processing and computer vision. Unlike traditional approaches such as RNNs or Convolutional Neural Networks Transformers heavily utilize the attention mechanism that has a structure that computes the interaction between each part of the input with other parts.

Accuracy and F-1 Score are the evaluation metrics used for machine learning and deep learning techniques. The accuracy formula is: $(True\ Positive) / (Total\ number\ of\ predictions)$. The F1 score combines the precision and recall scores and demonstrates the balance between correct and incorrect predictions. The F1 formula is:

$$F1 = \frac{True\ Positive}{True\ Positive + \frac{1}{2}x(False\ Positve + False\ Negative)} \quad (1)$$

Furthermore, the confusion matrix is provided to facilitate the computation of various model parameters, including precision and recall for evaluating model performance from different perspectives.

## IV. RESULTS AND DISCUSSION

We have collected 1444 code smell samples containing 62750 lines of code from 11 GitHub repositories. The test was conducted with 5 smell types. All code can be found in https://github.com/alinizam/BAP_Code_Analysis repository.

The results in Tables II and III demonstrate that DNN-based methods perform best with around 80% accuracy. In addition, cosine similarity provides a slightly lower accuracy compared to other implemented methods.

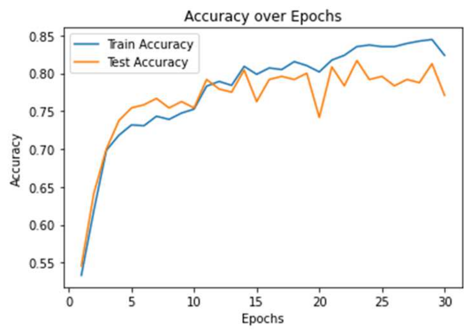
TABLE II. TEST RESULTS - SMELLS WITHOUT VARIABLE NAMES

| **Methods** | Cosine Similarity | kNN | LSTM | Transformers |
|---|---|---|---|---|
| Accuracy | 0.74 | 0.76 | 0.81 | 0.80 |
| F1 Score | 0.69 | 0.76 | 0.80 | 0.80 |

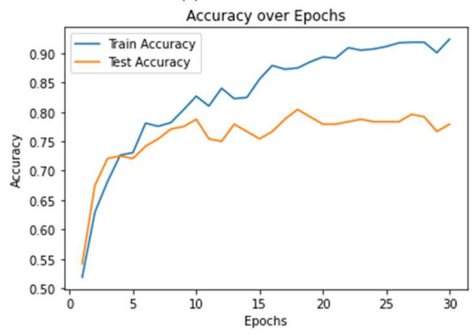
TABLE III. TEST RESULTS - SMELLS WITH VARIABLE NAMES

| **Methods** | Cosine Similarity | kNN | LSTM | Transformers |
|---|---|---|---|---|
| Accuracy | 0.75 | 0.78 | 0.79 | 0.80 |
| F1 Score | 0.69 | 0.77 | 0.80 | 0.80 |

Another inference of results is that there isn't any significant difference in results when comparing code smell AST with or without variable names.



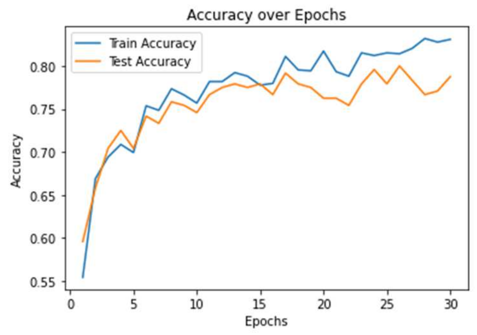
(a) Variable name is not used



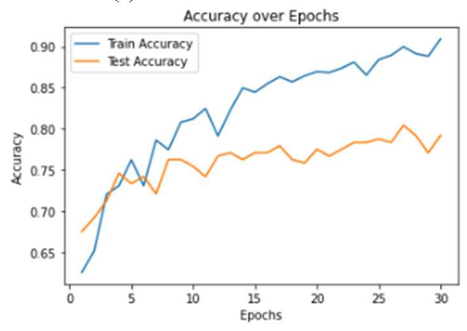
(a) Variable name is used

Fig. 2. LSTM accuracy over epochs graphs on the dataset

LSTM and transformers accuracy over epoch values have been given in Fig 2 a-b and 3a-b. They do not show a serious difference comparing the evaluation metrics correspondence with variable name usage in the learning process. The transformers model has a faster learning rate with parameter names. However, stopping the learning process at an early stage can be prone to overfitting learning to stop at some point.



(a) Variable name is not used



(b) Variable name is used

Fig. 3. Transformers accuracy over epochs graphs on the dataset

The confusion matrix demonstrates that a significant majority of our methods' predictions are correct as shown in Fig 4. It also demonstrates that our methods' prediction accuracy is lower when predicting the "Remove" type of smells. Because "Refactor" type smells sometimes also require or advise the movement of some code.

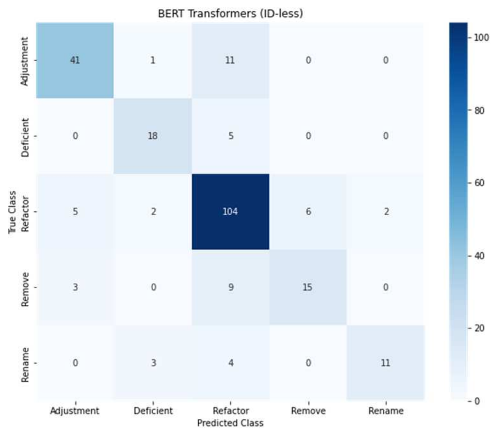


Fig. 4. The confusion matrix

Due to the small size of the data, overfitting problems were encountered in smell types with identifier information. When examining the confusion matrix, it was observed that the LSTM and Transformers methods made the most accurate predictions.

Although DNN networks had similar accuracy values, they performed better in unlabeled smell types. In smells with identifiers, increasing the number of epochs led to an increase in the overfitting problem, but this effect was minimized in smell types without identifiers.

On the other hand, the cosine similarity test had the lowest performance and consumed the most time. The reason behind the low performance of cosine similarity is that cosine similarity is based on a direct comparison between examples and has no system-level learning process. Although kNN performed better than Cosine Similarity, overfitting issues were still observed. In addition, we consider that the large size (dimensionality) of code embedding vectors reduced the effectiveness of KNN performance and decrease the performance. As seen in the confusion matrices, more than half of the dataset was labeled as refactor smell type.

## V. CONCLUSION

The principal contribution of this study is an inter-procedural model for detecting code smells. The accuracy and F1 scores ranged from 0.75 to 0.80. This result indicates that the proposed system will contribute to writing cleaner and more efficient code in the future. Another implication from the results is that the performance of machine learning-based methods is lower than DNN-based methods. Thus, DNN-based methods give better performance in understanding the structure of the code.

The main limitation of our study is working with a small number of code types and the relatively low number of code examples as the tool used requires compiling all the code and makes the data collection process difficult. Working with more code examples can improve system performance and the generalizability of findings.

Further research is needed to improve and enhance the code analysis capability of DNN systems to a comparable or higher level with metric-based systems. The other important research is to develop an embedding model representing the structure of the code. Another important topic is the evaluation of the different DNN method usages in code analysis.

## Acknowledgment

This work has been supported by Fatih Sultan Mehmet Vakif University Research Projects Coordination Unit under grant number 2022B1410D.

## References

[1] G. Lacerda, F. Petrillo, M. Pimenta, and Y. G. Guéhéneuc, "Code smells and refactoring: A tertiary systematic review of challenges and observations," J. Syst. Softw., vol. 167, 2020, doi: 10.1016/j.jss.2020.110610.

[2] N. Tsantalis and A. Chatzigeorgiou, "Ranking refactoring suggestions based on historical volatility," Proc. Eur. Conf. Softw. Maint. Reengineering, CSMR, pp. 25–34, 2011, doi: 10.1109/CSMR.2011.7.

[3] M. A. S. Bigonha, K. Ferreira, P. Souza, B. Sousa, M. Januário, and D. Lima, "The usefulness of software metric thresholds for detection of bad smells and fault prediction," Inf. Softw. Technol., vol. 115, no. March 2018, pp. 79–92, 2019, doi: 10.1016/j.infsof.2019.08.005.

[4] P. He, B. Li, X. Liu, J. Chen, and Y. Ma, "An empirical study on software defect prediction with a simplified metric set," Inf. Softw. Technol., vol. 59, pp. 170–190, 2015, doi: 10.1016/j.infsof.2014.11.006.

[5] K. Shi, Y. Lu, J. Chang, and Z. Wei, "PathPair2Vec: An AST path pair-based code representation method for defect prediction," J. Comput. Lang., vol. 59, no. May, p. 100979, 2020, doi: 10.1016/j.cola.2020.100979.

[6] T. Sharma, V. Efstathiou, P. Louridas, and D. Spinellis, "Code smell detection by deep direct-learning and transfer-learning," J. Syst. Softw., vol. 176, no. 110936, 2021.

[7] A. Alazba and H. Aljamaan, "Code smell detection using feature selection and stacking ensemble: An empirical investigation," Inf. Softw. Technol., vol. 138, no. August 2020, p. 106648, 2021, doi: 10.1016/j.infsof.2021.106648.

[8] Y. Sui, X. Cheng, G. Zhang, and H. Wang, "Flow2Vec: Value-flow-based precise code embedding," Proc. ACM Program. Lang., vol. 4, no. OOPSLA, 2020, doi: 10.1145/3428301.

[9] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "Code2Vec: Learning Distributed Representations of Code," Proc. ACM Program. Lang., vol. 3, no. POPL, pp. 1–29, 2019, doi: 10.1145/3290353.

[10] H. Liang, L. Sun, M. Wang, and Y. Yang, "Deep Learning with Customized Abstract Syntax Tree for Bug Localization," IEEE Access, vol. 7, pp. 116309–116320, 2019, doi: 10.1109/ACCESS.2019.2936948.

[11] X. Chen, U. C. Berkeley, and D. Song, "Tree-to-tree Neural Networks for Program Translation," no. Nips, 2018.

[12] "SonarQube." https://www.sonarsource.com/products/sonarqube/.

[13] M. Maddeh, S. Ayouni, S. Alyahya, and F. Hajjej, "Decision tree-based Design Defects Detection," IEEE Access, vol. 9, pp. 71606–71614, 2021, doi: 10.1109/ACCESS.2021.3078724.

[14] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, "A Survey of Machine Learning for Big Code and Naturalness," ACM Comput. Surv, vol. 51, no. 4, 2018.

[15] F. Khomh, S. Vaucher, Y. G. Guéehéeneuc, and H. Sahraoui, "A bayesian approach for the detection of code and design smells," Proc. - Int. Conf. Qual. Softw., pp. 305–314, 2009, doi: 10.1109/QSIC.2009.47.

[16] S. Bryton, F. Brito E Abreu, and M. Monteiro, "Reducing subjectivity in code smells detection: Experimenting with the Long Method," Proc. - 7th Int. Conf. Qual. Inf. Commun. Technol. QUATIC 2010, no. 3, pp. 337–342, 2010, doi: 10.1109/QUATIC.2010.60.

[17] F. Arcelli Fontana, M. V. Mäntylä, M. Zanoni, and A. Marino, "Comparing and experimenting machine learning techniques for code smell detection," Empir. Softw. Eng., vol. 21, no. 3, pp. 1143–1191, 2016, doi: 10.1007/s10664-015-9378-4.

[18] T. H. M. Le, H. A. O. Chen, and M. A. L. I. Babar, "Deep Learning for Source Code Modeling and Generation: Models, Applications and Challenges," pp. 1–37, 2014.

[19] Y. Zhuang, S. Suneja, V. Thost, G. Domeniconi, A. Morari, and J. Laredo, "Software Vulnerability Detection via Deep Learning over Disaggregated Code Graph Representation," arXiv Prepr., 2019.

[20] H. Liu, Z. Xu, and Y. Zou, "Deep learning based feature envy detection," ASE 2018 - Proc. 33rd ACM/IEEE Int. Conf. Autom. Softw. Eng., pp. 385–396, 2018, doi: 10.1145/3238147.3238166.

[21] I. Goodfellow, Y. Bengio, and A. Courville, Deep learning. MIT press, 2016.

[22] S. Hochreiter and J. Schmidhuber, "Long Short-Term Memory," Neural Comput., vol. 1780, pp. 1735–1780, 1997.

[23] E. Yahav and O. Levy, "Code2Seq:Generatıng Sequences From Structured Representatıons Of Code," in ICLR 2019 CODE2SEQ:, 2019, no. 1, pp. 1–19.

[24] H. Yonai, Y. Hayase, and H. Kitagawa, "Mercem: Method Name Recommendation Based on Call Graph Embedding," Proc. - Asia-Pacific Softw. Eng. Conf. APSEC, vol. 2019-Decem, pp. 134–141, 2019, doi: 10.1109/APSEC48747.2019.00027.

[25] R. Ferenc, Z. Tóth, G. Ladányi, I. Siket, and T. Gyimóthy, "A public unified bug dataset for Java," ACM Int. Conf. Proceeding Ser., pp. 12–21, 2018, doi: 10.1145/3273934.3273936.

[26] M. Pradel and K. Sen, "A Replication of 'DeepBugs: A Learning Approach to Name-based Bug Detection,'" in ACMProgram. Lang. 2, OOPSLA, 2018, vol. 2, no. November, doi: https://doi.org/10.1145/3276517.

[27] S. Benton, A. Ghanbari, and L. Zhang, "Defexts: A curated dataset of reproducible real-world bugs for modern JVM languages," Proc. - 2019 IEEE/ACM 41st Int. Conf. Softw. Eng. Companion, ICSE-Companion 2019, pp. 47–50, 2019, doi: 10.1109/ICSE-Companion.2019.00035.

[28] S. Habchi, N. Moha, and R. Rouvoy, "Android code smells: From introduction to refactoring," J. Syst. Softw., vol. 177, p. 110964, 2021, doi: 10.1016/j.jss.2021.110964.

[29] R. Shin et al., "Synthetic Datasets for Neural Program Synthesis," in ICLR, 2019, pp. 1–16.

[30] J. Devlin, M. W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," NAACL HLT 2019 - 2019 Conf. North Am. Chapter Assoc. Comput. Linguist. Hum. Lang. Technol. - Proc. Conf., vol. 1, no. Mlm, pp. 4171–4186, 2019.

[31] A. Vaswani et al., "Attention is all you need," Adv. Neural Inf. Process. Syst., vol. 2017-Decem, no. Nips, pp. 5999–6009, 2017.