Realizing Refactoring Prediction through Deep Learning

Lucas Rafael Rodrigues Pereira
Fedaral University of Lavras - UFLA
Lavras, Brazil

Dilson Lucas Pereira
Fedaral University of Lavras - UFLA
Lavras, Brazil

Rafael Serapilha Durelli
Fedaral University of Lavras - UFLA
Lavras, Brazil

ABSTRACT

Refactoring is the process of changing the internal structure of a software in order to improve its quality, without modifying its behavior. Recent studies have shown that the act of refactoring brings positive results for maintaining and understanding the code and the system as a whole. It turns out that, currently, this method is still little used, with expertise and intuition being the main factors that determine the need for software refactoring. Before starting the refactoring process, an analysis is essential to check whether refactoring is really necessary. Therefore, the present study analyzes artificial intelligence techniques, such as Deep Learning, to predict when software refactoring is essential. Deep Learning models like CNN, RNN, LSTM and DenseLayer were analyzed and compared using precision, recall and accuracy metrics. The results demonstrated that Machine Learning models performed better than Deep Learning algorithms using the same data set, however, the good performance of Deep Learning models stands out in scenarios where the data is very unbalanced.

KEYWORDS

Refactoring, Deep Learning, Machine Learning, Software Engineering

1 INTRODUCTION

Over the years, studies have established a correlation between refactoring operations and software quality [2, 17, 20]. Deciding when and what to refactor poses the biggest challenge for developers as it marks the initiation of the refactoring process [22]. Software development teams often perceive refactoring as technical debt, and like any other change, it comes with associated costs. These costs can be even more significant when the size or impact of the refactoring is unknown, as delaying it may lead to more extensive impacts caused by the "defective" code [22].

To aid with the refactoring process and minimize costs, developers have been employing static analysis tools like Sonarqube [28], PMD [4], and ESLint [25][10]. However, these tools primarily rely on strategies based on well-known and cataloged code smells, such as God Class and Long Method. While code smells serve as sufficient motivation for refactoring, they are not the sole reason for it. Consequently, the strategy of relying solely on static and heuristic analysis tools may not be as effective [16].

The tools utilized for static analysis often exhibit a considerable number of false positives when providing refactoring recommendations, leading to a loss of confidence among developers [12]. Although these tools offer some degree of customization, the complexity of modern software, encompassing a vast array of classes, frameworks, and packages, makes it likely that false positives will still arise [12].

The application of Deep Learning in the realm of Software Engineering has gained significant relevance in recent years. This has led to the development of novel models aimed at addressing diverse software quality challenges, defining architectures, modularization, and tackling specific software issues [6].

Researchers have been exploring the use of Artificial Intelligence (AI) approaches to recommend refactoring points, employing techniques such as search algorithms [23, 27], pattern mining [9], and machine learning [5]. However, when employing Deep Learning as a Machine Learning technique, there is an expectation of obtaining favorable results in predicting software code refactoring, particularly when applied across different areas of software engineering, such as predicting defects [13], code understanding [21], and code smells [7].

Recent years have witnessed an increasing number of studies exploring the potential of machine learning and deep learning to enhance and facilitate the refactoring process. Within this context, the primary objective of this paper is to determine the feasibility of predicting the need for refactoring through a Deep Learning model. We have chosen to direct our efforts based on the following proposition: "***Can a Deep Learning model anticipate software code refactoring?***". To address this inquiry, the study was partitioned into three sub-questions. The first sub-question is:

RQ_1 - "How to transform software codes into valid input for a Deep Learning model?"

To address this challenge, two source code representation models were evaluated: CODEBERT[14] and Code2Vec[3]. Among the two models, CODEBERT[14] was chosen for its capability to transform software codes into vectors and assign weights to these vectors based on the selected programming language. Moreover, it offers pre-trained models in JAVA. By leveraging CODEBERT[14], instead of directly inputting raw software code to the model, a more feasible input is provided, enabling easier training. This is particularly advantageous as it allows the application of algorithms such as oversampling, undersampling, CNN, among others, that specifically operate with numerical vectors.

The second sub-question is:

RQ_2 - "Which Deep Learning model will perform best in predicting refactoring in software code?"

By employing pre-processing techniques to address class imbalance and leveraging neural networks, Long Short Term Memory(LSTM), Batch normalization, and Pooling, we were able to attain an accuracy of 69% and a precision of 70%. Such results can be deemed optimistic, especially in an unbalanced test data scenario.

The third sub-question is:

RQ_3 - "How does the Deep Learning model perform compared to machine learning models"

To assess the outcomes of the Deep Learning model in comparison to machine learning models, we selected a study [5] that employs the same dataset and presents results for various machine learning algorithms. The main contributions of this work are threefold: (*i*) Development of a methodology for predicting Extract Method using Deep Learning; (*ii*) Evaluation and comparison of different Deep Learning models to ascertain their effectiveness; (*iii*) Establishment of a groundwork for future research in other types of refactoring or different software languages.

2 BACKGROUND

2.1 Refactoring

Code Refactoring or just refactoring is the practice of modifying software to enhance its internal structure while preserving its external behavior. A paramount reason to prioritize code refactoring, particularly during the development cycle, is to elevate code quality, mitigate the occurrence of defects, and enhance scalability and maintainability for future code maintenance.

Nonetheless, determining the extent and specific areas to refactor poses a challenge for many developers. Engaging in refactoring activities incurs additional costs, as it necessitates the involvement of experienced developers to thoroughly analyze the code and understand the entire context of the application. Unfortunately, such efforts are rarely compensated and are often considered as technical debt [18, 19].

The significance of refactoring becomes evident in the time saved when implementing code changes. By refactoring and organizing the code during development, developers spend less time deciphering and understanding the code. Consequently, the investment made in refactoring or organization processes is recouped in both the short and long term [18].

The identification and removal of code smells in the code can be automatically suggested by a tool. However, recent studies delve into the application of Machine Learning, search-based algorithms, or heuristics [8] to discover refactoring opportunities. Given the increasing popularity of using Machine Learning algorithms to identify refactoring points in software, this study specifically focuses on applying the relatively less explored Deep Learning technique to evaluate its effectiveness in detecting refactoring points in software code.

2.2 Deep Learning based refactoring

The ability of machine learning models to interpret code was made possible with innovative approaches like CODEBERT[14], which transforms software code into numerical vectors for assessment by artificial intelligence models.

In a comprehensive literature review conducted by Naik et al. [24], the analysis covers 17 studies focusing on the use of Machine Learning and Deep Learning techniques for software code refactoring. The review delves into the techniques employed by each study, the programming languages used for training, as well as model evaluation and pre-processing techniques. From this investigation, it was concluded that recurrent neural networks (RNN), convolutional neural networks (CNN), multilayer perceptrons (MLP), and graph neural networks (GNN) emerged as the most widely used deep learning models, with MLP showing the most promising performance.

Moreover, the study by Naik et al. [24] reveals that the majority of these studies are predominantly based on JAVA, focus on method-level refactorings, and use only one programming language. These characteristics closely align with the model developed in this work.

3 APPROACH

The model comprises three main phases: Data Collection, Training & Testing, and Model Evaluation. Each of these phases will be outlined briefly here and then explored in further detail in dedicated subsections.

The data collection phase focuses on selecting the dataset, mining the chosen repositories, and pre-processing the data. Upon detecting a refactoring of type "Extract Method," the methods involved in the commit are gathered, and each method is labeled to indicate the presence or absence of the "Extract Method" refactoring. These labeled methods are then converted into binary tokens (0,1) and assigned pre-trained weights using CodeBERT[14].

Subsequently, the training phase commences, where the pre-processed data is trained and tested using various Deep Learning algorithms. The size and number of Deep Learning layers are adjusted based on the algorithm used, such as RNN, CNN, and LSTM, to achieve optimal results for each model.

Finally, the model's outcomes are collected, and the accuracy of each model is assessed. For each model, precision, accuracy, and recall are evaluated. To evaluate the overall model performance, the results from Machine Learning[5] algorithms, as well as results obtained from developers, are also considered.

3.1 Experimental Sample

In order to establish a meaningful comparison with the machine learning algorithms, we adopted the same sampling strategy as used by [5]. This involved sampling a large number of Java projects from three different sources: (*i*) Apache Software Foundation (ASF): ASF is an organization that hosts and supports a wide range of Apache projects. The repository used in this study contains 860 Java-based projects; (*ii*) F-Droid: F-Droid is a repository for Android code, hosting a collection of 1352 open-source mobile app projects; (*iii*) GitHub: For GitHub it was planned to use the first 10000 repositories containing only JAVA code, but the tools were able to successfully collect only 9072.

The combined dataset from the three repositories provides a diverse range of software projects, exhibiting significant variations in terms of code quantity, size, complexity, domains, technologies used, and development teams involved. This diversity ensures that the dataset is representative of real-world software projects, making the results of the study more applicable and generalizable.

3.2 Refactoring Extraction

The data collection process was conducted using the Refactoring Miner tool [29]. This tool operates by cloning the repositories selected from the designated repositories list. It then examines the history of the master branch, detecting various types of refactoring present in the commits. For each cloned repository, Refactoring

Miner accesses the master branch and analyzes all commits, starting from the oldest to the newest.

The tool conducts a pairwise analysis of commits, accurately determining, with 98% accuracy, the types of refactoring that occurred between these two commits. Upon detecting a refactoring of the EXTRACT METHOD type, the code before the refactoring is stored, along with a label indicating the type of refactoring. The final outcome of this data collection process is a table containing multiple methods found in the repositories. One column of the table contains these methods, while the other column includes binary markers (0s and 1s) indicating the presence or absence of the method extraction refactoring. This dataset forms the foundation for the subsequent stages of the research.

After the data collection, a cleaning process was performed to address issues such as duplicate codes and methods with conflicting labels. During the collection process, a method that was selected for containing a refactoring in one commit might be marked as having no refactoring when examined in another commit. To prevent redundancies during the training process and to avoid confusion in the learning process, all duplicate codes were removed from the database. Additionally, duplicate codes with different labels were excluded, except for a single copy containing a refactoring label, to ensure data consistency and accuracy in the subsequent stages of the research.

3.3 Tokenization of source code

Once the methods have been collected and properly classified, it is still necessary to transform the data into a valid input for the Deep Learning models. Source code and files in natural language require treatment so that they can be used by Deep Learning algorithms that only use real numbers.

The decision to use CODEBERT[14] for transforming source code into numerical vectors was made to take advantage of the possibility of using pre-trained data, saving time and labor costs. CODEBERT[14] utilizes the BERT[11] architecture, which employs transformers in an encoder-decoder architecture with an attention mechanism for natural language pre-training. The transformers in BERT[11] are bi-directional, enabling context extraction from words in both left-to-right and right-to-left directions. This approach allows CODEBERT to efficiently convert natural language, including source code, into numeric arrays for use in Deep Learning algorithms.

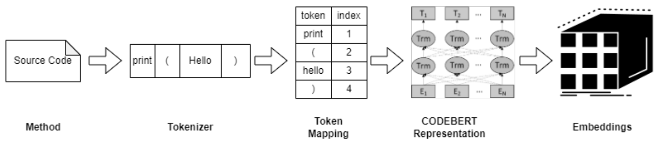

Figure 1: embedding process utilizing CODEBERT

3.4 Model Training

To train the Deep Learning model for predicting whether a method should undergo EXTRACT METHOD type refactoring, a dataset containing real examples of methods that underwent refactoring and examples without refactoring was used.

The choice of the EXTRACT METHOD type for this study is due to its popularity among others refactorings, as it contains a larger number of instances, which is essential for improving the performance of Deep Learning algorithms.

The following four algorithms were used to train and evaluate the performance of the models:

- CNN: Convolutional neural networks are widely used in training with images and computer vision, but because they deal well with binary classification data, it is believed to have relevance for training [26].
- RNN: Recurrent neural networks, designed to handle sequential data such as text and software code. They have loops in their structure that allow relevant information to be passed on [15].
- Dense layers: Dense layer is the basic layer in deep neural networks. In it, all neurons are connected to all neurons in the previous layer and the next layer and generate an output through an activation function.
- LSTM: Variation of recurrent neural networks (RNN) adapted to handle very long sequences and retain information longer than a simple recurrent network [15].

To address the natural data imbalance issue, the simple undersample algorithm was employed. This technique balanced the dataset by selecting an equal number of methods classified as "without refactoring" to match the number of methods classified as "with refactoring," resulting in a 50% balance on each side.

After balancing the data, the hyperparameters of each algorithm were fine-tuned, and the number of layers was adjusted based on the complexity of the algorithm and its performance to achieve satisfactory results. Once the models were trained, the prediction for a given method would be the probability of whether it should or should not undergo refactoring, providing valuable insights to developers for code improvement.

3.5 Evaluation

To address the research questions posed in this paper, the mean precision, accuracy, and recall of each of the four algorithms were compared, considering different configurations and hyperparameterization. The performance of each Deep Learning model was assessed in comparison with others, to identify the most effective one, and then compared with Machine Learning models from the literature [5]. Furthermore, the results obtained from the models were compared with a survey involving real developers to gain additional insights into the model's performance. This comprehensive evaluation aimed to provide a robust understanding of the predictive capabilities of the Deep Learning models for software code refactoring.

The work by [5] already provides the precision, accuracy, and recall of the models, making it unnecessary to recalculate these measures for comparison purposes. By leveraging the existing results from that study, we can efficiently compare the performance of our Deep Learning models with the Machine Learning models, providing valuable insights into the effectiveness of our approach for predicting software code refactoring.

3.6 Implementation and execution

All the refactoring instances collected by RefactoringMiner [29] were stored in a local MySQL database. To extract the methods from the commits, a Python script was developed to detect and separate the methods efficiently. This script helped in organizing and preprocessing the data for further analysis and training of the Deep Learning models.

The data collection algorithm, executed on a machine with 16GB of RAM, a Quadcore CPU with 3.2GHz of processing, and a dedicated SSD, took an average of 16 minutes per repository to collect data from all established repositories. However, it is worth noting that around one-tenth of the repositories took more than an hour to run, leading to occasional connection errors between MySQL and JAVA or Python during the process. Despite these challenges, the data collection was successfully completed, allowing for further analysis and training of the Deep Learning models.

For training the Deep Learning models, Google Collab Pro was utilized, which provided a virtual machine with 50GB of RAM, an adjustable CPU, and 200GB of disk space. The training process for each of the algorithms, consisting of 200 epochs, took approximately 24 hours to complete. The use of Google Collab Pro enabled efficient and resourceful training of the models, contributing to the successful evaluation and comparison of their performance.

4 RESULTS

All the questions have been addressed and answered herein.

RQ$_1$ - "How to transform software codes into valid input for a Deep Learning model?"

After implementing and analyzing the Deep Learning models using the Code2Vec and CODEBERT source code interpretation algorithms, we obtained the results presented in Tables 1 and 2. These tables display the mean precision, accuracy, and recall values for each model, along with their respective hyperparameter configurations. The results offer insights into the models' performance in predicting software code refactoring, specifically the EXTRACT METHOD type.

The comparison between the models using different algorithms sheds light on their effectiveness in detecting potential refactoring points in the software code. These findings contribute to addressing the research questions and demonstrate the potential of Deep Learning techniques for automating refactoring recommendations.

Code2Vec	Precision	Recall	Accuracy
CNN	22.00	13.47	89.59
RNN	21.32	11.02	89.95
LSTM	71.72	00.96	52.93
Dense Layers	16.96	14.56	89.95

Table 1: Precision, Recall and Accuracy of Deep Learning models trained using Code2Vec

CODEBERT	Precision	Recall	Accuracy
CNN	56.40	80.09	58.64
RNN	52.23	93.82	53.49
LSTM	66.59	59.48	65.17
Dense Layers	65.68	58.21	64.72

Table 2: Precision, Recall and Accuracy of Deep Learning models trained using CODEBERT

	Precision	Recall	Accuracy
CNN	56.40	80.09	58.64
RNN	52.23	93.82	53.49
LSTM	66.59	59.48	65.17
Dense Layer	65.68	58.21	64.72
Average	59.97	72.90	60.50

Table 3: Precision, Recall and Accuracy of Deep Learning models trained with the same dataset

Observation 1: CODEBERT performed better than Cod2Vec in the four Deep Learning models. The observed results indicate that CODEBERT[14] performed better compared to Code2Vec. As much as the accuracy values of Code2Vec exceed the values generated by CODEBERT, the precision and the recall indicate a better prediction rate of CODEBERT.

Observation 2: CODEBERT was the model chosen to interpret the source codes used in this work. In addition to the superior performance of Code2Vec, CODEBERT does not require local training like Code2Vec, which makes the replication of the experiment more reliable, such as reducing the cost of processing the project.

RQ$_2$ - "Which Deep Learning model will perform best in predicting refactoring in software code?"

The table 3 displays the precision, recall and accuracy obtained in each of the proposed Deep Learning models.

Observation 3: LSTM was the model with the best result among the proposed models. LSTM achieved an average accuracy of 65.17%, precision of 66.59% and recall of 59.48% during the training of the datasets. The second best performing model was Dense Layers with results of 64.72% in accuracy, 65.68% in precision and 64.72% in recall when trained with the same data.

Observation 4: RNN and CNN present much higher recall than precision. The RNN and CNN models presented, respectively, a recall of 93.82% and 80.09% when trained with the same database. However, the accuracy of the models was 52.23% for RNN and 56.40% for CNN. The LSTM and Dense Layers models had superior accuracy and precision, but had a much lower recall when compared to the RNN and CNN models.

Observation 5: All models managed to perform well in relation to the natural imbalance of the data. Although the proportion of non-refactored methods was almost 90% in relation to the number of refactoring methods with method extraction, the accuracy of all models remained above 52.23%, with a minimum recall of 53.49%.

RQ$_3$ - "How does the Deep Learning model perform compared to machine learning models"

To compare the performance of the Deep Learning models with the Machine Learning models, we will compare the results obtained with the work of Aniche et al., [5], who analyzed and mapped the results of 6 different Machine Learning algorithms. The results can be checked in Table 4

	Precision	Recall	Accuracy
Logistic Regression	0.80	0.87	0.82
SVM	0.77	0.88	0.80
Naive Bayes(gaussian)	0.65	0.95	0.70
Decision Tree	0.81	0.86	0.82
Random Forest	0.80	0.92	0.84
Neural Network	0.84	0.84	0.84
Average	0.77	0.88	0.80

Table 4: Precision, Recall and Accuracy of Machine Learning models

In Table 5 we can see the comparison between the mean accuracy, precision and recall of the Deep Learning and Machine Learning models. All models use the same GitHub, Apache and F-droid database, which makes comparison possible.

	Precision	Recall	Accuracy
Average Deep Learning Models	0.59	0.72	0.60
Average Machine Learning Models	0.77	0.88	0.80
Difference	-0.18	-0.16	-0.20

Table 5: Comparison between Deep Learning and Machine Learning models

Observation 6: Machine Learning models demonstrated better performance compared to Deep Learning models. As all models share the same database, it is possible to compare the results, however, it is believed that with a larger and more diverse database the Deep Learning algorithms can improve the results.

In order to bring another comparison of the results found in relation to the performance of the models, 10 JAVA developers were consulted and asked if a certain method should undergo refactoring. The methods in question were extracted from the database used to train the models. The developers' response can be analyzed in the following Table 6

The level of experience of the developers can be seen in Figure 2 and the question form used can be found in the domain: [1].

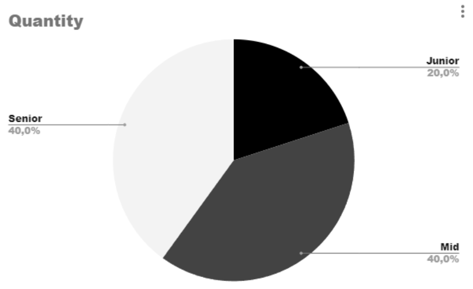

Figure 2: Level of expertise of developers who participated in the survey

	Method 1	Method 2	Method 3	Method 4	Method 5	Precision
Dev 1	1	0	1	0	1	0.6
Dev 2	1	1	1	1	1	1.0
Dev 3	1	0	0	1	1	0.6
Dev 4	1	0	1	1	1	0.8
Dev 5	1	0	0	1	1	0.6
Dev 6	1	0	1	1	1	0.8
Dev 7	1	0	0	0	0	0.2
Dev 8	1	1	0	1	0	0.6
Dev 9	1	0	1	1	1	0.8
Dev 10	1	0	1	1	1	0.8
Precision	1.0	0.2	0.6	0.8	0.8	

Table 6: Developer success rate when predicting a refactoring analyzed by Deep Learning models

Observation 7: Developers have a success rate similar to that of Deep Learning models. The average success rate of developers with 5 methods taken from the dataset used to train the Deep Learning models obtained a difference of approximately 8%, as shown in Table 7

	Precision
Average Deep Learning Models	0.5997
Average Developers	0.6800
Difference	-0.0803

Table 7: Comparison between the accuracy of Deep Learning models with the average accuracy of developers

5 THREATS TO VALIDITY

5.1 Construction Validity

During the process of collecting the repositories, the use of the RefactoringMiner tool was mentioned[29]. This tool has an accuracy of 98% and a recall of 87%, for the use of EXTRACT METHODS, so when collecting the repositories we do not re-calculate these values based on the collected data. In order to compare results with Machine Learning algorithms, we chose to use the same repositories used by [5]. However, the extraction process is costly and can lead to difficulties for those mining the same repository instances. Approximately 8% of the repositories failed to be collected, but this number may vary according to the resources allocated for extraction, such as the tools used to mine the datasets.

5.2 Internal Validity

The natural imbalance of the EXTRACT METHOD class presents many more instances of non-refactored classes than of refactored classes in a ratio of 9 to 1. To deal with the imbalance, data balancing algorithms were used without creating generic values, such as Random Undersampler. Knowing that pre-processing the data can lead to less accurate models, the performance of the model can vary when compared to a scenario based on reality. The order of refactorings was not considered for this study. It is believed that the current state of the method should be sufficient for assessing the need for EXTRACT METHOD refactoring. By taking into account only the body of the method to analyze Deep Learning models, relevant information external to the method may be lost when doing the refactoring. Future work should take this aspect into account.

5.3 External Validity

Results are based on Open Source projects, which may affect generalizability in an industrial context. It is necessary, however, to replicate this study for a dataset with projects actually in use by the industry and with a greater range of domains. By choosing JAVA as the object of study, the results found may not be replicable for other programming languages. It is believed that the results may be similar for any object-oriented language such as JAVA, however future work should take this fact into account.

6 CONCLUSION

This research work focuses on the importance of software refactoring and proposes a Deep Learning model, using CODEBERT as a basis, to predict when a method should be refactored through the EXTRACT METHOD. The study compares Deep Learning models and assesses their performance against Machine Learning models. While the Deep Learning models achieved favorable results, the Machine Learning models were more effective for predicting EXTRACT METHOD refactorings in the same dataset. Nevertheless, the research provides valuable insights for predicting refactoring in software code and suggests avenues for further exploration in the realm of Deep Learning and software quality. Future work can expand the scope to include other types of refactoring and explore different approaches to leverage Deep Learning effectively in predicting refactorings and addressing software quality issues.

REFERENCES

[1] [n. d.]. Formulário de validação de modelos DL. https://forms.gle/pL9HRfZnJmocQwgt6. Acessado: 2021-06-11.

[2] Eman Abdullah AlOmar, Mohamed Wiem Mkaouer, Ali Ouni, and Marouane Kessentini. 2019. Do design metrics capture developers perception of quality? an empirical study on self-affirmed refactoring activities. *arXiv preprint arXiv:1907.04797* (2019).

[3] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–29.

[4] Juan Martín Sotuyo Dodero Clément Fournier Pelisse Romain Robert Sösemann Andreas Dangel, BBG. 2022. https://github.com/pmd/pmd.

[5] Mauricio Aniche, Erick Maziero, Rafael Durelli, and Vinicius Durelli. 2020. The effectiveness of supervised machine learning algorithms in predicting software refactoring. *IEEE Transactions on Software Engineering* (2020).

[6] Anders Arpteg, Björn Brinne, Luka Crnkovic-Friis, and Jan Bosch. 2018. Software engineering challenges of deep learning. In *2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE, 50–59.

[7] Muhammad Ilyas Azeem, Fabio Palomba, Lin Shi, and Qing Wang. 2019. Machine learning techniques for code smell detection: A systematic literature review and meta-analysis. *Information and Software Technology* 108 (2019), 115–138.

[8] Abdulrahman Ahmed Bobakr Baqais and Mohammad Alshayeb. 2020. Automatic software refactoring: a systematic literature review. *Software Quality Journal* 28, 2 (2020), 459–502.

[9] Gabriele Bavota, Andrea De Lucia, Andrian Marcus, and Rocco Oliveto. 2014. Recommending refactoring operations in large software systems. In *Recommendation Systems in Software Engineering*. Springer, 387–419.

[10] Moritz Beller, Radjino Bholanath, Shane McIntosh, and Andy Zaidman. 2016. Analyzing the state of static analysis: A large-scale evaluation in open source software. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 1. IEEE, 470–481.

[11] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[12] Lisa Nguyen Quang Do, James R. Wright, and Karim Ali. 2022. Why Do Software Developers Use Static Analysis Tools? A User-Centered Study of Developer Needs and Motivations. *IEEE Transactions on Software Engineering* 48, 3 (2022), 835–847. https://doi.org/10.1109/TSE.2020.3004525

[13] Marco D'Ambros, Michele Lanza, and Romain Robbes. 2012. Evaluating defect prediction approaches: a benchmark and an extensive comparison. *Empirical Software Engineering* 17, 4 (2012), 531–577.

[14] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155* (2020).

[15] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.

[16] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. 2013. Why don't software developers use static analysis tools to find bugs?. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 672–681.

[17] Yoshio Kataoka, Takeo Imai, Hiroki Andou, and Tetsuji Fukaya. 2002. A quantitative evaluation of maintainability enhancement by refactoring. In *International Conference on Software Maintenance, 2002. Proceedings.* IEEE, 576–585.

[18] Miryung Kim, Thomas Zimmermann, and Nachiappan Nagappan. 2012. A field study of refactoring challenges and benefits. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. 1–11.

[19] Philippe Kruchten, Robert L Nord, and Ipek Ozkaya. 2012. Technical debt: From metaphor to theory and practice. *Ieee software* 29, 6 (2012), 18–21.

[20] Robert Leitch and Eleni Stroulia. 2004. Assessing the maintainability benefits of design restructuring using dependency analysis. In *Proceedings. 5th International Workshop on Enterprise Networking and Computing in Healthcare Industry (IEEE Cat. No. 03EX717)*. IEEE, 309–322.

[21] Kui Liu, Dongsun Kim, Tegawendé F Bissyandé, Taeyoung Kim, Kisub Kim, Anil Koyuncu, Suntae Kim, and Yves Le Traon. 2019. Learning to spot and refactor inconsistent method names. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 1–12.

[22] Mateus Lopes and Andre Hora. 2022. How and why we end up with complex methods: a multi-language study. *Empirical Software Engineering* 27, 5 (2022), 1–42.

[23] Thainá Mariani and Silvia Regina Vergilio. 2017. A systematic review on search-based refactoring. *Information and Software Technology* 83 (2017), 14–34.

[24] Purnima Naik, Salomi Nelaballi, Venkata Sai Pusuluri, and Dae-Kyoo Kim. 2023. Deep Learning-Based Code Refactoring: A Review of Current Knowledge. *Journal of Computer Information Systems* (2023), 1–15.

[25] Milos Djermanovic Nicholas C. Zakas, Brandon Mills. 2022. https://eslint.org/.

[26] Keiron O'Shea and Ryan Nash. 2015. An introduction to convolutional neural networks. *arXiv preprint arXiv:1511.08458* (2015).

[27] Mark O'Keeffe and Mel O Cinnéide. 2008. Search-based refactoring for software maintenance. *Journal of Systems and Software* 81, 4 (2008), 502–516.

[28] SonarSource S.A. 2008–2022. https://sonarqube.org.

[29] Nikolaos Tsantalis, Ameya Ketkar, and Danny Dig. 2020. RefactoringMiner 2.0. *IEEE Transactions on Software Engineering* (2020), 21 pages. https://doi.org/10.1109/TSE.2020.3007722