Detection Bad Code Smells By Using Deep Machine Learning Approaches

1st Hiba M. Yahya
Software Department
College of Computer Sciences & Mathematics , University of Mosul
Mosul, Iraq
hibamoneer@uomosul.edu.iq

2nd Dujan B. Taha
Cyber Security Department
College of Computer Sciences & Mathematics , University of Mosul
Mosul, Iraq
dujan_taha@uomosul.edu.iq

2023 1st International Conference on Advanced Engineering and Technologies (ICONNIC) | 979-8-3503-0648-4/23/$31.00 ©2023 IEEE | DOI: 10.1109/ICONNIC59854.2023.10467672

Abstract - Code smells are an indication of deviation from design principles or implementation in the source code. Early detection of these code smells increases software quality. In contemporary times, diverse artificial neural network models are employed in software engineering to detect code smells without the need for source code access, but only large training sets. The aim of a study is to present a Recurrent Neural Network, Long Short Term Memory and Gated Recurrent Units models for detecting three types of code smells (Feature Envy, Long Method, and God Class) based on Java projects. Three code smell datasets were considered. Various performance metrics were employed for comparing the results. The experimental findings suggest that the proposed models demonstrate superior performance in identifying code smells, and augmenting the training data can improve the accuracy of predictions.

Keywords - Refactoring, Bad Code Smell, Deep Machine Learning.

I. Introduction

Any changes made to the source code that violate software design rules can result in code smells. Design issues or developer modifications that could potentially affect the quality of the system in the future are referred to as code smells and cause problems during maintenance. If code smells are not addressed , they may result in the deterioration of software projects and create technical debt. Therefore, code smells can serve as an indicator to identify whether the source code requires [1]. The identification of poor code smells in the source code is critical in initiating the code refactoring process. Generally, techniques for detecting code smells depend on object-oriented metrics as inputs to determine the presence of such smells in software projects. To identify and resolve issues in the source code, numerous various static analysis tools and code restructuring techniques have been devised that scrutinize the source code[2].

Machine learning approaches involve the training of supervised models using data extracted from the same or a different software project. To model the source code components, metrics are used, similar to heuristic-based approaches. However, ML approaches differ in that they do not necessitate the specification of threshold values. Instead, they depend on on data-driven learning to determine whether a particular code component is categorized as "smelly" or "non-smelly".

Supervised learning algorithms,such as recurrent neural networks (RNNs) , have been responsible for the remarkable progress in deep learning in recent years. RNNs are currently active in various practical applications like text generation, auto-translation, speech recognition, and code smell detection [3] .

The primary objective of this research is to introduce a deep convolutional neural network models for detecting three different code smells and evaluate its performance based on various performance metrics, including recall, precision, accuracy, and F-measure. The paper is structured as follows : Section two discusses the related literature, section three provides a background on detection strategies for code smells and deep machine learning . Methodology of our research is presented in Section four . Section five presents the experimental results and subsequent discussions, followed byconclusions in the final section.

II. RELATED WORK

After 1999, when Fowler et al.[1] published a book that outlined various bad code smells and the corresponding refactoring techniques, research into detecting these code smells began in the field. Numerous literature reviews and surveys have been carried out in the domain of code smell identification and refactoring [4][5][6][7].These investigations have demonstrated several methods and tactics for identifying poor code smells in contemporary software systems through the utilization of machine learning methods [8].

Romeo L. [3] A prototype utilizing neural networks, machine learning, and deep learning for code smell detection was developed and implemented using the Python programming language. Subedi [9] suggested a method to collect, process and analyze code smells of different open-source projects and detect code smells in an intelligent way using the LSTM machine learning model. Sharma et al.[10] used CNN and RNN as their major hidden layers along with auto encoder model . They perform training and assessment on C# examples and Java code. Mhawish et al.[11] proposed was an approach for predicting code smells using machine learning techniques and software metrics, which incorporated the Local Interpretable Model-Agnostic Explanations (LIME) algorithm to improve comprehension of the machine learning model's decision-making process, and to identify the specific features that have an impact on the prediction model's decisions.

III. BACKGROUND

Previous studies have proposed several techniques and approaches for identifying poor code smells. This section provides a concise overview of the pertinent information regarding detection strategies for code smells and deep machine learning .

- **Code Smells Detection :**

Code smell detection is typically created on a grouping of object-oriented metrics and predefined threshold value , aimed at identifying the main indications that define the code smells [12] . A variety of detection approaches rely on heuristics and detection rules that compare metric values obtained from source code with empirically established thresholds, in order to differentiate between code artifacts affected by a particular type of smell and those that are not. The choice of appropriate threshold values is crucial to the performance of detectors since it strongly influences their effectiveness. Hence, identifying suitable typical thresholds is a crucial factor in developing effective detection strategies. Table (1) showcases the detectors employed in creating code smell datasets

Smells	**Detectors**
God Class	iPlasma , PMd
Data Class	iPlasma , Fluid Tool , Antipattern Scanner
Feature Envy	iPlasma , Fluid tool
Long Method	iPlasma , PMD , Marinescu[2]

Table (1) Detectors considered for building code smell datasets

The following three typical code smells that will be used in the research:

1- **God class:** is an anti-pattern in software design where a single class has too much responsibility and becomes overly complex. It tends to make the code difficult to maintain and modify. Such a class often contains excessive code, multiple methods, and tightly coupled dependencies, leading to high coupling and low cohesion.

2- **Feature envy:** is a code smell that occurs when a method is overly focused on the data of another object instead of its own. This happens when a method excessively accesses and manipulates the fields or methods of another object. This practice goes against the principle of encapsulation, which can result in low cohesion and tight coupling in the code.

3- **Long method:** a code smell that talks about a method that contains too many lines of code. So it make code harder to read, maintain and understand. Therefore long methods should be refactored into smaller, more manageable methods.

- **Deep Machine Learning :**

Deep machine learning is a subset of machine learning that is belong to artificial intelligence that contains the use of complex algorithms and neural networks to analyze and learn from large sets of data. The word "deep" refers to using multiple layers of neural networks in the process of a data, more accurate and sophisticated models will be created . By using this technology the field of machine learning will be revolutionize , enabling computers to recognize patterns, classify data, and make predictions with greater accuracy than ever before. Many applications belong to deep machine learning like natural language processing, image recognition, and predictive analytics [13]. RNN are a type of deep neural networks specifically designed for processing sequential data by preserving contextual information from previous inputs. Unlike traditional feed forward neural networks that process inputs independently, However, there is a problem appears in RNNs named vanishing gradients, where the gradients used to update the network's parameters become too small and cause the network to stop learning. To address this matter, Various variations of RNNs have been suggested, counting Gated Recurrent Units (GRUs) and Long Short-Term Memory (LSTM) , which employ additional mechanisms to better regulate the flow of information through the network. LSTM, or Long Short-Term Memory, is a type of RNN architecture used in deep learning. LSTM networks aim to tackle the vanishing gradient problem of traditional RNNs by introducing specialized units known as memory cells. Each memory cell is composed of three gates : the input ,output , and forget gates. The flow of information will be device by these gates into and out of the memory cell, as well as determine what information is important to keep or forget. The flow of new input into the memory cell is regulated by the input gate, while the output gate controls the output of the memory cell to the next layer of the network. The forget gate plays a crucial role in deciding which information should be discarded from the memory cell [14][15].

IV. Method

Various machine learning methods have been developed for identifying code smells. The data is typically divided into two separate sets: a training and testing sets. During the training phase the model is trained using training set, while to assess the model's performance the testing set is employed. After the model is built, it's important to evaluate its effectiveness. In this study, a proposed method involves comparing the performance for detecting code smells of two deep convolutional network models, which is illustrated in Fig 1.

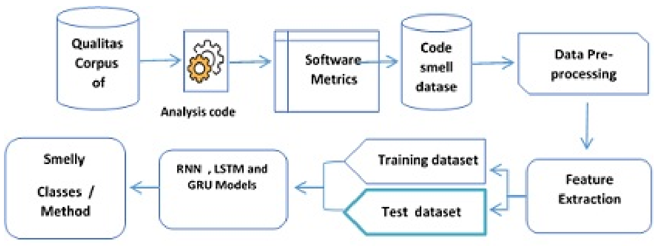


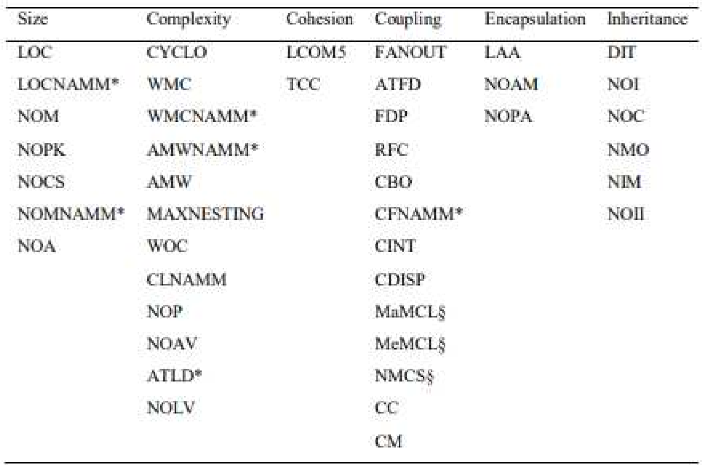
Fig 1. Proposed process for code smells detection

V. Collection and Modeling of Data

The research adopts a supervised learning approach that depend on a vast array of software metrics as input variables. This approach forms the foundation for training neural network models and facilitates generality of the findings. To conduct the analysis and experimentation, a significant number of systems or datasets need to be available. For this study, this model employed the suggested datasets in Fontana et al. [8]. Authors nominated 74 open-source systems from the Qualitas Corpus, which is a collection of 111 Java systems be appropriate to various application domains and described by varying sizes, collected by Tempero et al. [16].

VI. Software Metrics

Software metrics are crucial in evaluating and improving software quality. They are used to measure and describe different aspects of software engineering products. The fundamental aim of software metrics is to assess and quantify particular system features , such as inheritance, encapsulation , and classes. Software metrics can be classified into several types, such as those for source code examination, software testing, and quality assurance [17] . For this study, the selected metrics are at the project , package , method , and class level, and are listed in Tabel 2.



Size	Complexity	Cohesion	Coupling	Encapsulation	Inheritance
LOC	CYCLO	LCOM5	FANOUT	LAA	DIT
LOCNAMM*	WMC	TCC	ATFD	NOAM	NOI
NOM	WMCNAMM*		FDP	NOPA	NOC
NOPK	AMWNAMM*		RFC		NMO
NOCS	AMW		CBO		NIM
NOMNAMM*	MAXNESTING		CFNAMM*		NOII
NOA	WOC		CINT		
	CLNAMM		CDISP		
	NOP		MaMCL§		
	NOAV		MeMCL§		
	ATLD*		NMCS§		
	NOLV		CC		
			CM		

Table 2. selected metrics for the project , package , method , and class level

VII. Data Pre-processing and feature extraction

The first step in data analysis is data pre-processing, which involves converting raw data into a more suitable and usable format for analysis. It contain cleaning the data by removing missing values or outliers, transforming the data by scaling or normalizing it. It is critical because it helps to enhance the quality and effectiveness of data analysis, leading to more accurate and reliable results. While , feature extraction is involves selecting a subset of features from the original set of variables that can best capture the patterns and relationships present in the data. it commonly used in machine learning and pattern recognition tasks, where it helps to reduce the dimensionality of the data and increase the efficiency and accuracy of the analysis.

VIII. The evaluation of three models

To assess the model's effectiveness, the study utilized standard performance metrics built on the confusion matrix, such as recall,precision,f-measure and accuracy.The utilization of a confusion matrix is to evaluate the classification performance of a model, by using a predetermined set of test data , with each row representing a predicted class and each column corresponding to an actual class. Tables (3,4,5) show the performance analysis for three models and tables (6,7,8) show the Error metrics for the three models.

The performance Metrics.	**Precision**	**Recall**	**Accuracy**	**F1_score**
God Class	**100%**	**96%**	**97%**	**98%**
Feature Envy	**97%**	**100%**	**97%**	**98%**
Long Method	**95%**	**98%**	**94%**	**97%**

Table 3. The performance metrics for LSTM model

The performance Metrics.	**Precision**	**Recall**	**Accuracy**	**F1_score**
God Class	100%	92%	95%	96%
Feature Envy	95%	95%	92%	95%
Long Method	92%	96%	91%	94%

Table 4. The performance metrics for RNN model

The performance Metrics.	**Precision**	**Recall**	**Accuracy**	**F1_score**
God Class	90%	100%	90%	95%
Feature Envy	95%	98%	94%	97%
Long Method	**90%**	100%	90%	94%

Table 5. The performance metrics for GRU model

The Error Met.	**MAE**	**RMSE**	**RAE**	**RRSE**
God Class	**0.025**	**0.158**	**0.076**	**0.486**
Feature Envy	**0.028**	**0.167**	**0.181**	**1.083**
Long Method	**0.050**	**0.237**	**0.307**	**1.296**

Table 6. The Error metrics for LSTM model

The Error Met.	**MAE**	**RMSE**	**RAE**	**RRSE**
God Class	0.050	0.225	0.153	0.683
Feature Envy	0.084	0.290	0.428	1.534
Long Method	0.091	0.302	0.464	1.535

Table 7. The Error metrics for RNN model

The Error Met.	**MAE**	**RMSE**	**RAE**	**RRSE**
God Class	0.1	0.316	1	3.162
Feature Envy	0.056	0.237	0.363	1.532
Long Method	0.098	0.313	0.538	1.714

Table 8. The Error metrics for GRU model

IX. The discussion of a result

Tables (3,4,5) shows the performance metrics of three different types of recurrent neural network models LSTM, RNN, and GRU, on three different types of code smells (God Class, Feature Envy, and Long Method). The metrics evaluated are precision, recall, accuracy, and F1-score.

Precision indicates the proportion of true positives out of all the examples classified as positive. Recall refers to the proportion of true positives out of all the actual positives. Accuracy refers to the proportion of correct classifications out of all examples. A good metric to use when both precision and recall are important is the F1 score.

When examining the tables, we can see that the LSTM model is the best, so it generally outperforms the RNN and GRU models across all three code smells in terms of precision, recall, accuracy, and F1-score. We notice that the LSTM model has a precision of 100% for God Class, whereas the precision values of the RNN and GRU models have 100% and 90%, respectively. Similarly, for Feature Envy, the LSTM model has a precision of 97%, while the precision values of the RNN and GRU models have 95% and 95%, respectively.

According to the specific evaluation metrics that used in this study, the LSTM model seems to be the most efficient in detecting these types of code smells in software. It is significant to bear in mind, that these outcomes are derived from a particular dataset and evaluation methodology and may not apply to all scenarios.

The results of the experiments using LSTM, RNN and GRU on the God class, Feature envy, and Long method datasets reveal that all models are effective in identifying and classifying instances with a high degree of accuracy. In the case of the God class dataset, all models achieved a high accuracy rate of 97% , 95% and 90%, respectively, indicating that they can correctly identify instances of this class. However, the precision and recall values were quite high at 100% for LSTM and RNN models, suggesting that there were instances where the models struggled to correctly classify certain instances.

For the Feature envy dataset, the precision and recall values were high for all models, with LSTM achieving 97% ,RNN achieving 92% and GRU achieving 95%, respectively. This indicates that the models were able to correctly identify instances of Feature envy with a high degree of accuracy. In the case of the Long method dataset, all models performed well with high accuracy and F1-score values. LSTM achieved a precision and recall of 95% and 98%, respectively, RNN achieved a precision and recall of 92% and 96%, respectively. while GRU achieved a precision and recall of 90% and 100%, respectively This suggests that all models are effective in identifying and classifying instances of Long method. However, similar to the God class dataset, the precision and recall values were quite low for RNN.

The error metrics were calculated to evaluate the accuracy of the predictions made by the models including MAE, RMSE, RAE, and RRSE. Overall, the error metrics suggest that all models were able to make relatively accurate predictions, with low values of MAE and RMSE indicating a low level of prediction error. However, the RAE and RRSE values were relatively high for some datasets, indicating that there may be instances where the models struggle to make accurate predictions.

X. Conclusion

In conclusion, the results of the experiments suggest that LSTM ,RNN and GRU models can be effective in identifying and classifying instances of God class, Feature envy, and Long method models for identifying bad code smells in Java projects are compared based on software metrics. with a high degree of accuracy. However. The suggested detection system aims to identify three specific code smells in Java projects. To assess the performance of the models, experiments were conducted using different numbers of hidden layers and epochs. The experimental results were evaluated and built on performance measures such as recall, precision, accuracy, and F-Measure. The findings indicate that the LSTM model has high accuracy in all the datasets used and the potential in detecting code smells. This is due to the fact that LSTM models can capture long-term dependencies and can effectively model sequential data, which is essential in analyzing code smells. In contrast, RNN models suffer from the vanishing gradient problem and struggle to capture long-term dependencies, resulting in less accurate predictions. Thus, the use of LSTM models in detecting bad code smells is a promising approach that can potentially improve the quality and maintainability of software projects Additionally, a correlation between software metrics and code smells is uncovered by the study.

References :

[1] M. Fowler *et al.*, “Refactoring Improving the Design of Existing Code Second Edition,” 2019.

[2] A. Kaur and G. Dhiman, *A review on search-based tools and techniques to identify bad code smells in object-oriented systems*, vol. 741, no. September. Springer Singapore, 2019. doi: 10.1007/978-981-13-0761-4_86.

[3] RomeoLQuoiJr, “Deep Learning-Based Code Smell Detection Using CNN and RNN,” Computer Science & Technology, 2020.

[4] M. I. Azeem, F. Palomba, L. Shi, and Q. Wang, “Machine learning techniques for code smell detection: A systematic literature review and meta-analysis,” *Inf. Softw. Technol.*, vol. 108, pp. 115–138, 2019, doi: 10.1016/j.infsof.2018.12.009.

[5] T. Sharma *et al.*, “A Survey on Machine Learning Techniques for Source Code Analysis,” vol. 0, no. 0, 2021, [Online]. Available: http://arxiv.org/abs/2110.09610

[6] H. M. Yahya and D. B. Taha, “Software Code Refactoring : A Comprehensive Review,” vol. 2023, pp. 71–80, 2023, doi: 10.33899/edusj.2023.137163.1298.

[7] 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, no. April, 2020, doi: 10.1016/j.jss.2020.110610.

[8] F. Arcelli Fontana and M. Zanoni, “Code smell severity classification using machine learning techniques,” *Knowledge-Based Syst.*, vol. 128, pp. 43–58, 2017, doi: 10.1016/j.knosys.2017.04.014.

[9] S. Subedi, “INTELLIGENT CODE. SMELL. DETECTION SYSTEM USING DEEP LEARNING,” 2021.

[10] T. Sharma, V. Efstathiou, P. Louridas, and D. Spinellis, “Code smell detection by deep direct-learning and transfer-learning,” *J. Syst. Softw.*, vol. 176, 2021, doi: 10.1016/j.jss.2021.110936.

[11] M. Y. Mhawish and M. Gupta, “Predicting Code Smells and Analysis of Predictions: Using Machine Learning Techniques and Software Metrics,” *J. Comput. Sci. Technol.*, vol. 35, no. 6, pp. 1428–1445, 2020, doi: 10.1007/s11390-020-0323-7.

[12] U. Mansoor, M. Kessentini, B. R. Maxim, and K. Deb, “Multi-objective code-smells detection using good and bad design examples,” *Softw. Qual. J.*, vol. 25, no. 2, pp. 529–552, 2017, doi: 10.1007/s11219-016-9309-7.

[13] M. A. Wani, F. A. Bhat, S. Afzal, and A. I. Khan, *Advances in Deep Learning*, vol. 57, no. January. 2019. doi: 10.1007/978-981-13-6794-6.

[14] and A. J. S. Aston Zhang, Zachary C. Lipton, Mu Li, “Dive into DeepLearning,” p. 987, 2020.

[15] A. Aksoy, Y. E. Ertürk, S. Erdoğan, E. Eyduran, and M. M. Tariq, “Estimation of honey production in beekeeping enterprises from eastern part of Turkey through some data mining algorithms,” *Pak. J. Zool.*, vol. 50, no. 6, pp. 2199–2207, 2018, doi: 10.17582/journal.pjz/2018.50.6.2199.2207.

[16] E. Tempero *et al.*, “The Qualitas Corpus: A curated collection of Java code for empirical studies,” *Proc. - Asia-Pacific Softw. Eng. Conf. APSEC*, pp. 336–345, 2010, doi: 10.1109/APSEC.2010.46.

[17] N. A. A. Khleel and K. Nehéz, “Deep convolutional neural network model for bad code smells detection based on oversampling method,” *Indones. J. Electr. Eng. Comput. Sci.*, vol. 26, no. 3, pp. 1725–1735, 2022, doi: 10.11591/ijeecs.v26.i3.pp1725-1735.

[18] 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.