STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LXVIII**, Number 2, 2023
DOI: 10.24193/subbi.2023.2.02

SOFTWARE MAINTAINABILITY AND REFACTORINGS PREDICTION BASED ON TECHNICAL DEBT ISSUES

L. BERCIU AND V. MOLDOVAN

Abstract. Software maintainability is a crucial factor impacting cost, time and resource allocation for software development. Code refactorings greatly enhance code quality, readability, understandability and extensibility. Hence, accurate prediction methods for both maintainability and refactorings are vital for long-term project sustainability and success, offering substantial benefits to the software community as a whole. This article focuses on prediction of software maintainability and the number of needed code refactorings using technical debt data. Two approaches were explored, one compressing technical debt issues per software component and employing machine learning algorithms such as ExtraTrees, Random Forest, Decision Trees, which all obtained a high accuracy and performance. The second approach retained multiple debt issue entries and utilized a Recurrent Neural Network, although less effectively. In addition to the prediction of the requisite number of code refactorings and software maintainability for individual software components, a comprehensive analysis of technical debt issues was conducted before and after the refactoring process. The outcomes of this study contribute to the advancement of a dependable prediction system for maintainability and refactorings, presenting potential advantages to the software community in effectively managing maintenance resources. From all the employed models, the ExtraTrees model yielded the most optimal predictive outcomes. To the best of our knowledge no other approaches of using ML techniques for this problem have been reported in the literarture.

1. Introduction

In the last decades, software has known a continuous growth facing increasingly demanding expectations and requirements. Consequently, the software development process must aim for optimal efficiency. The objective is to create software systems that are bug-free, easily modifiable and updatable, and

Received by the editors: 19 June 2023.

2010 *Mathematics Subject Classification.* 68N99.

1998 *CR Categories and Descriptors.* D2.0 [**Software Engineering**]: General – *Standards*; D2.9 [**Software Engineering**]: Management – *Software Quality Assurance*.

Key words and phrases. Software Quality, Sonarqube, Refactoring, Code Smells.

capable of accommodating new features seamlessly. The maintenance phase of software development is a substantial part of its life cycle. This phase is of utmost importance as it spans the entire lifespan of the software product, commencing immediately after the completion of the development process. Numerous studies have emphasized the significance of software maintainability [9], [5], [21].

There is a strong connection between software systems' maintainability and technical debt issues. There are several types of software issues that can be detected by performing a static analysis of the software systems. Without actually running the code, software static analysis looks for potential problems and enhances code quality. It specifically looks at a program's source code to find potential problems such syntax mistakes, security flaws, performance problems, and maintainability concerns. A small number of issues implies a higher maintainability and vice versa. One of the code's problems found in the static analysis process is represented by the technical debt.

As presented in [2], [16], and [27], there is a strong connection between the software systems' detected issues, its maintainability and the refactorings performed on that software. Refactoring aims to lower technical debt while increasing readability, maintainability, and scalability of the software system. When the maintainability is too low and there are too many issues which overcomplicate the development process, a refactor is needed. After successfully and correctly performing the refactor, the number of issues should decrease and the software maintainability should increase, improving the quality of the software.

With the growing interest in the field of software development, there has been a significant rise in the development of tools aimed at enhancing the software development process. These tools offer various capabilities, including the ability to measure code metrics, conduct static analysis to identify technical debt issues, and provide suggestions for code improvements. Among the widely recognized tools in this domain are SonarQube [1], PyLint [2], ESLint [3], cppcheck [4], CheckStyle [5], FindBugs [6]. There are also tools that aim to detect

[1]Sonarqube. https://www.sonarqube.org/.

[2]PyLint, https://pypi.org/project/pylint/

[3]ESLint, https://eslint.org/

[4]cppcheck, https://cppchecksolutions.com/

[5]CheckStyle, https://github.com/checkstyle/checkstyle

[6]FindBugs, https://spotbugs.github.io/

code refactorings between different code versions, such as RefactoringMiner [7], RefactoringCrawler [8], RefDiff [9], and several others.

The goal of this paper is to predict the number of refactorings that need to be performed, and based on this number to classify the maintainability of each software component, aiming to obtain a performance as high as possible. To accomplish this, multiple intelligent algorithms have been employed and analyzed to identify the most suitable algorithm for this specific task. In addition to predicting software maintainability and number of needed code refactorings, an analysis of the technical debt issues before and after the refactoring process has been conducted.

The problem of predicting software maintainability and number of needed refactorings based on technical debt issues was addressed firstly as a classification problem, and then as a regression problem. The classification task referred to classifying each software component into one of the following maintainability classes: "Great", "Good", and "Poor", while the regression task referred to predicting the number of needed refactorings by each software component.

The subsequent sections of this paper are organized as follows: Section 2 outlines the related work, Section 3.1 presents the data preparation, and Section 3.2 expounds on the architectural configurations of the employed machine learning algorithms. A comparative analysis is presented in Section 4, followed by a more comprehensive examination of the SonarQube issues in Section 5. Ultimately, Section 6 offers the drawn conclusions and points towards potential directions for future work.

2. Related work

Ensuring the correctness and efficiency of modern software systems is essential. Software maintainability, being a crucial quality factor, plays a vital role in ensuring the long-term sustainability of software systems and mitigating technical debt. It facilitates effective bug fixing, enables seamless software updates and improvements, fosters collaboration within development teams, and supports the overall adaptability and evolution of software systems. By prioritizing maintainability, organizations can streamline their software development processes, enhance productivity, and deliver reliable, high-quality software products that meet the evolving needs of their customers and stakeholders.

The topic of software maintainability prediction has received a significant interest, leading to numerous studies dedicated to addressing this problem.

[7]RefactoringMiner, https://github.com/tsantalis/RefactoringMiner
[8]RefactoringCrawler, http://dig.cs.illinois.edu/tools/RefactoringCrawler/
[9]RefDiff, https://github.com/aserg-ufmg/RefDiff

A comprehensive review study conducted by Elmidaoui et al. [8] examined 77 research studies published between 2000 and 2018 that aimed to predict software maintainability based on various software quality metrics. The review presented in [8] provides a detailed analysis of the employed maintainability prediction techniques, validation methods, accuracy criteria, overall accuracy of machine learning (ML) techniques, and the techniques offering the best performance.

In [26], Van Koten and Gray found that ML techniques, including Bayesian networks [23], outperformed regression-based models in prediction accuracy. The study showed that ANNs [24] capture complex non-linear relationships and approximate any measurable function, SVM/R [6], [7] excels in learning classification and regression tasks, especially with high-dimensional data, DT [4] offer a straightforward and comprehensible approach and FNF methods [13] handle limited or missing data, while RA [20] is a simple and reliable technique, particularly useful with multiple independent variables. This represented a reason for which, in this approach, machine learning algorithms were chosen to be used in the detriment of regression-based models.

The prediction techniques utilized in these studies can be broadly classified into two main groups: ML techniques and statistical techniques. Statistical techniques encompassed various approaches, such as regression analysis (RA), probability density function (PD), Gaussian mixture model (GMM), discriminant analysis (DA), weighted functions (WF), and stochastic model (SM). In contrast, ML techniques encompassed artificial neural networks (ANN), case-based reasoning (CBR), regression and decision trees (DT), Bayesian networks (BN), evolutionary algorithms (EA), support vector machine and regression (SVM/R), fuzzy and neuro fuzzy (FNF), inductive rule-based (IRB), ensemble methods (EM), and clustering methods (CM).

The review study [8] showed that the statistical techniques, more popular from 2000 until 2007, are only effective when a linear or predetermined relationship exists between the dependent and independent variables. With the advent of ML techniques, researchers started exploring both statistical and ML approaches to assess their predictive capabilities for maintainability. In [14], Kaur and Kaur emphasized that traditional parametric statistical data analysis methods may be insufficient and suggested that the utilization of ML algorithms or pattern recognition approaches, which are inherently nonparametric, could lead to improved prediction accuracies.

The process of refactoring holds an important significance owing to its capacity to enhance code quality, improve maintainability, and foster collaboration among developers. By eliminating code smells, mitigating technical debt, and optimizing code performance, refactoring contributes to the development of robust and scalable software systems. Consequently, numerous research studies have been conducted to comprehensively analyze the relationship between refactoring and software maintainability, as well as to explore predictive methods for anticipating the need for refactorings.

In [12], Hegedus et al. presents an enhanced dataset comprising verified refactoring data pertaining to open-source systems. The study reveals that refactoring is frequently employed on entities exhibiting low maintainability, signifying developers' proactive efforts to address deteriorated code. Metrics associated with size, complexity, and coupling exhibit notable increases in refactored elements, indicating developers' focus on improving these aspects. However, the analysis suggests that metrics related to code clones have a comparatively lesser impact.

In [1], an investigation on the prediction of software refactoring by employing Support Vector Machine (SVM) and optimization algorithms is presented, exploring the relationship between code coverage and the effectiveness of the test suite in an evolutionary context. The authors examine the application of SVM in conjunction with genetic algorithms to forecast refactoring at the class level. Utilizing a dataset derived from open-source software systems, the study achieves promising levels of accuracy, ranging from 84% to 93%. The performance is further enhanced by integrating SVM with the optimization algorithms.

3. Experiment and study plan

The experiment was designed in the following manner: firstly, the data was gathered, then the dataset was prepared The last included several steps: Preparing the technical debt dataset: computing the number of issues per software component, then associating numerical values to severity (BLOCKER: 5, CRITICAL:4, MAJOR:3, MINOR:2, INFO:1) and type (CODE SMELL:1, BUG:2, VULNERABILITY:3), removing the N/A entries and finally computing for each software component the mean of the severity, debt and type values; Preparing the refactoring dataset implied computing the number of refactorings per software component. Finally, the dataset was created after merging the technical debt dataset with the refactoring dataset. After that, data was split into training (70%) and testing (30%). The models were trained, and then the testing data was used to evaluate them. The last step was represented by analyzing the obtained results and drawing conclusions.

3.1. **Data Preparation.**

In order to predict the software maintainability and number of needed refactorings based on technical debt issues, several steps need to be performed. The first element needed is represented by data. In this research investigation, a comprehensive analysis was conducted on three open-source Java projects, namely jEdit [10], FreeMind [11], and TuxGuitar [12]. The study encompassed two distinct categories of data pertaining to these projects. Firstly, the technical debt issues encountered in jEdit version 5.5, FreeMind version 1.0.1, and TuxGuitar version 1.5.2 were examined. Secondly, the refactorings performed between these versions and subsequent versions of each project, specifically jEdit 5.6, FreeMind 1.1.0, and TuxGuitar 1.5.3, were taken into account. The dataset [17] containing all technical debt issues information about the jEdit project version 5.5, FreeMind project version 1.0.1 and TuxGuitar project version 1.5.2, has been previously used in other studies as well, such as [18].

The data needed for the Technical Debt analysis step of the experiment was collected by running SonarQube on the three projects and extracting the issues data found by the tool, such that the details provided by the tool will be considered as attributes. While the tool execution for jEdit and FreeMind was straightforward, namely successfully compiling the projects and running the Sonarqube tool by using the Sonar Scanner version matching the build system used (ANT in both cases), the data collection for TuxGuitar proved to be more difficult, as TuxGuitar is divided in a multitude of individual projects compiled by Maven build system, more exactly 65 individual projects. From those, we selected the base TuxGuitar project, TuxGuitar Android Resource, TuxGuitar AudioUnit, TuxGuitar CoreAudio, TuxGuitar Editor Utils, TuxGuitar GM Utils, TuxGuitar Lib and TuxGuitar UI toolkit. SonarQube was executed on each individual project and then the issues fetched and merged together.

In this research study, a selective approach was adopted regarding the attributes of the technical debt issues under consideration. After running the chosen static analysis tool, a report was obtained that contained a list of all detected issues, each issue associated with the name of the software component in which it's located, and several other attributes such as severity, debt, type, creation date, rule, update date, and others. Specifically, the severity, debt, and type of each issue were thoroughly investigated. This decision was based on the notion that certain attributes, such as the key, did not provide significant or pertinent information for the classification or regression model. Furthermore, some attributes required more intricate examination and preprocessing, which warranted their inclusion in future research endeavors. In addition to the severity, debt, and type of each issue, the computation of the number of issues per component was performed and subsequently utilized in the prediction process.

[10]jEdit, http://www.jedit.org/
[11]FreeMind, https://freemind.sourceforge.net/
[12]TuxGuitar, https://sourceforge.net/projects/tuxguitar/

Regarding the final representation of the technical debt issues data, two distinct approaches were pursued:

- The first approach involved compressing the technical debt issues for each project into a single instance per software component. To obtain the final values of severity and type for each component, these attributes were mapped to numerical values using the following scheme: Severity = INFO: 1, MINOR: 2, MAJOR: 3, CRITICAL: 4, BLOCKER: 5 and Type = CODE SMELL: 1, BUG: 2, VULNERABILITY: 3. After the mapping process, the mean values of severity and type were computed by summing their respective values for each component and dividing the sum by the number of issues associated with that component. When calculating the mean for the debt attribute, instances with a value of N/A were excluded from consideration, and the associated issues were removed from the total count per component. By performing these operations, each software component was represented by a single instance.
- The second approach did not involve compressing the issue instances into a single instance per class/component. Instead, it focused on removing issues that had an N/A value for the debt attribute. This decision was made to enhance interpretability for the model, as N/A values posed difficulties in interpretation. Additionally, this approach also resulted in a decrease in the number of issues per component, which had been previously computed.

The data utilized in this study comprised information related to the refactorings conducted between two versions of each project, which was obtained through the employment of the RefactoringMiner tool. This tool facilitated a comparison between two project versions and generated a report detailing the refactorings executed during this transition. The provided refactoring information encompassed the type of each refactor, a concise description of its purpose, and the specific component on which the refactoring was performed. Additionally, the number of refactorings for each software component was computed. These attributes provided valuable insights into the code's condition and offered suggestions for improving its maintainability. To avoid challenges associated with high-dimensional data, such as overfitting, computational complexity, and data sparsity, only the count of performed refactorings per software component was considered in this study.

To create the final dataset, the technical debt issues data and the refactorings data were merged. The software components served as the common element between these datasets, enabling the addition of a new column in the technical debt issues dataset that represented the number of refactorings performed on each specific component. Consequently, the analyzed data fed into the intelligent algorithm contained information pertaining to the severity, debt, type, and count of technical debt issues for each software component, alongside the number of refactorings conducted on that particular component.

The problem in this study was initially formulated as a classification task and later as a regression task. In the regression setting, the predicted output was the estimated number of refactorings required for each component. In the classification problem, the output classes were defined as follows:

- "Great" category: Corresponded to components that had fewer than 5 refactorings performed on them.
- "Good" category: Associated with software components that underwent between 5 and 20 refactorings.
- "Poor" category: Assigned to software components that had more than 20 refactorings performed on them.

3.2. **Architectural Configurations of Employed Machine Learning Algorithms.**

Various architectural configurations were evaluated for the prediction of software maintainability and the required quantity of refactorings in the analyzed projects. Employing the initial dataset, which comprised a single entry for each software component, multiple models were trained and achieved commendable performance.

For the classification approach, the LazyClassifier from the *lazypredict.Supervised* library was employed, and several models were studied, including the Extra Tree Classifier [10], LGBM Classifier [15], Random Forest Classifier [3], and K-Neighbors Classifier [11]. Cross-validation with 5 folds and the *f1_macro* scoring metric were applied using the *cross_val_score* function from the *sklearn.model_selection module.* Additionally, the MLPClassifier [25] from the *sklearn.neural_network* module was evaluated with different configurations, such as varying the number of neurons (100, 150, and 200), considering activation functions like *Relu* and *logistic sigmoid*, and utilizing the $lbfgs$ and *adam* solvers as optimization methods. The results are presented in Table 1.

For the regression approach, the *LazyRegressor* from the *lazypredict.Supervised* library was utilized, and several models were examined, including the Extra Tree Regressor, K-Neighbor Regressor, Hist Gradient Boosting Regressor, Random Forest Regressor, and Decision Tree Regressor. Similar to the classification approach, cross-validation was conducted using a KFold of 5, and the negative mean squared error was used as the scoring metric. The multi-layer perceptron (MLP) was also applied for regression, with the number of neurons set to 20, 25, and 30, and the activation functions and solvers remaining the same as those used in the classification task.

Table 1. Results obtained for Lazy Classifier models and MLP and RNN models

Model	Accuracy	Recall	Precision	F1-Score
ExtraTreesClassifier	**0.92**	**0.92**	**0.92**	**0.92**
RandomForestClassifier	0.90	0.90	0.90	0.90
LGBMClassifier	0.88	0.88	0.88	0.88
DecisionTreeClassifier	0.88	0.88	0.88	0.88
ExtraTreeClassifier	0.82	0.81	0.83	0.82
KNeighborsClassifier	0.77	0.76	0.79	0.77
SVC	0.70	0.70	0.71	0.70
MLP	0.69	0.65	0.69	0.67
LogisticRegression	0.63	0.63	0.62	0.62
LinearSVC	0.60	0.59	0.58	0.58
Perceptron	0.56	0.56	0.56	0.56
RNN	0.57	0.58	0.56	0.56

The second dataset, which included multiple entries per component corresponding to detected technical debt issues, was used for training a recurrent neural network (RNN). The RNN model consisted of the following components:

- An Embedding layer with a length equal to the training data's length.
- A LSTM (long-short term memory) layer followed, utilizing Relu activation functions for both regression and classification tasks, and sigmoid activation function solely for classification.
- A Dropout layer with a dropout rate of 0.2.
- Another LSTM layer with the same activation functions as the previous LSTM layer
- A subsequent Dropout layer with a dropout rate of 0.2
- A Dense layer with Relu activation function for regression and classification, and sigmoid activation function only for classification,
- Another Dropout layer, identical to the previous two. This layer helps prevent overfitting and introduces noise, making the network more robust to dependencies on specific features.
- The last layer was a Dense layer with either 3 output channels for classification (corresponding to the defined maintainability classes) or 1 output channel for regression. The activation function used was Softmax for classification and no activation function for regression.

The model was compiled using the sparse categorical cross entropy loss function for the classification algorithm and mean squared error loss function for the regression algorithm. The optimizer used for both algorithms was adam. The results are presented in Table 2.

Table 2. Results obtained for Lazy Regressor models, MLP and RNN models

Model	R-Squared	Adjusted R-Squared	RMSE	MAE
ExtraTreesRegressor	**0.92**	**0.91**	**2.75**	**1.77**
RandomForestRegressor	0.90	0.89	3.12	2.36
ExtraTreeRegressor	0.81	0.81	2.88	2.54
DecisionTreeRegressor	0.88	0.88	3.01	1.87
LGBMRegressor	0.88	0.88	3.07	2.18
KNeighborsRegressor	0.75	0.75	2.87	1.99
SVR	0.65	0.65	3.88	3.28
LinearSVR	0.59	0.58	3.97	3.02
MLP	0.58	0.58	3.85	3.25
RNN	0.50	0.50	4.21	3.25
LinearRegression	0.49	0.48	4.69	3.88

4. Comparative analysis

In Table 1, the results obtained for LazyClassifier models, MLP and RNN models are presented. The best performing model was represented by ExtraTreesClassifier using the first approach of handling the data, while the lowest performance was obtained by the RNN model using the second data approach. The accuracy of the ExtraTreesClassifier was 0.92, also having the same value for the recall, precision and F1-Score metrics. This suggests that that the classification model is performing at a high level, making correct predictions, and effectively capturing positive cases. As for the RNN model, it achieved an accuracy of 0.57, a recall of 0.58 and a precision and a F1-Score of 0.56, indicating that model has some level of predictive ability, but the performance is quite low comparative to the other employed models. The RNN model might be making correct predictions for a portion of the data, but there are also instances where it's struggling to provide accurate results. This needs to be further investigate in order to be improved.

The regression results presented in Table 2 are similar to the ones from the classification task presented in Table 1, the ExtraTrees algorithms being the most performant one. The RNN model behaved better than LinearRegression, but its performance still needs to be further analysed and improved. The RNN model obtained a R-Squared and and Adjusted R-Squared of 0.50, suggesting that he model explains about 50% of the variability in the target variable, being a moderate fit. A RMSE of 4.21 implies that the model's predictions have an average error of around 4.21 units, while a MAE of 3.25 suggests that, on average, the model's predictions are off by about 3.25 units from the actual values.

Following the execution of the ExtraTrees classifier, the outcomes for each maintainability class are displayed in Table 3. The analysis reveals that the "Good" class achieved the highest performance, while conversely, the "Poor" class exhibited the lowest performance. This observation aligns with the characteristics of the initial dataset, which exhibited an imbalance prior to undergoing data augmentation. Specifically, the dataset contained a smaller number of instances classified as "Poor" compared to instances classified as "Great" and "Good."

Table 3. Results obtained for ExtraTrees model

Output Class	Accuracy	Recall	Precision	F1-Score
Great	0.92	0.91	0.93	0.92
Good	0.94	0.94	0.94	0.94
Poor	0.90	0.91	0.89	0.90
Total	0.92	0.92	0.92	0.92

The study focused on comparing the obtained results of software maintainability with the values of the maintainability index [22], which is a popular measurement method. The maintainability index categorizes software into three classes: Bad, Satisfactory, and Acceptable. The jEdit 5.5 project was classified as "Satisfactory," FreeMind 1.0.1 as "Bad," and TuxGuitar 1.5.2 as "Satisfactory" based on their maintainability index values. To perform a fair comparison between the results and the maintainability index, the mean value of the maintainability index for each project was computed.

The prediction model showed high performance based on the training data, with the lowest metrics observed for the "Poor" class. However, the observed maintainability index values for the three projects did not fully align with this finding. The mapping between software components and maintainability classes based on the number of needed refactorings did not match the mapping based on maintainability index values. This might suggest that there is no direct relationship between the number of refactorings or technical debt issues and the maintainability index value [19]. The discrepancy between the study's findings and the maintainability index can be attributed to the limitations of metrics used in the maintainability index, which do not fully capture the complexities of object-oriented software. Since the studied projects were developed using the object-oriented paradigm, it is expected that the results may not align perfectly with the maintainability index. Thus, our study confirms one more time that maintainability index does not accurately characterize object oriented systems.

5. A deeper dive into SonarQube issues

In the previous sections, we provided a high level overview on how maintainability index and technical debt metrics exhibit variation between two successive versions of a project's release timeline. On the Technical Debt experiment side, we leveraged the SonarQube issues and the refactorings data from the provided datasets, selected the data most suitable for the experiment from both sources, merged it together and fed it to artificial intelligence tools. While this proved to be effective in computing the results of the study and providing a general answer on the aforementioned research questions, we decided to further refine and improve our research findings by following a particular path: diving deeper into SonarQube issues.

Throughout this section, we will present the particularities of SonarQube issues found inside each project by doing a classification of their types, severities and numbers, see how they compare between versions and offer a final comparison with the initial results from the previous section.

5.1. **JEdit 5.5 and 5.6.** JEdit proved to have the highest number of issues found for both versions, from all projects under study. For version 5.5, we have extracted a total of 139.905 issues, from which 134.901 were labeled as CODE_SMELLS and 4004 were labeled as BUG. For Version 5.6, the number of total issues was 63899, from which 57773 were labeled as CODE_SMELL and 6126 were labeled as BUG. Interestingly, the latter version also reported 8 issues explicitly labeled as vulnerabilities. The sonar rules spanned multiple file types, such as .java, .html and .xml. The general data can be visualised in Table 4, while Table 5 shows the data for Java only files.

Table 4. General issues comparison between jEdit versions

Project	**CODE_SMELL**	**BUG**	**VULNERABILITY**
jEdit 5.5	135901.0	4004.0	-
jEdit 5.6	57773	6126	8

To properly show how the issues fluctuated between the two releases, tables 6 and 5.1 show how the percentages between issue types changed. We can

Table 5. Java issues comparison between jEdit versions

Project	CODE_SMELL	BUG	VULNERABILITY
jedit-5-5.csv	29093.0	441.0	-
jedit-5-6.csv	20355	541	8

observe a decrease in code smells from 97.14% to 90.40% when it comes to total code smells reported to the other issues and a staggering numeric decrease from 135901 to 57773. While this looks like an improvement on a first glance, if we take BUG issues into consideration, we can observe an actual increase from 2.86% to 9.59% between versions, more specifically, from 4004 to 6126 issues reported as bugs. An extra 8 vulnerabilities were also found. While the total number of issues may have decreased, we can clearly observe that their severity increased, as the number of bugs increased by 50% and bugs having a higher severity than code smells in general. We can conclude that, at least for now, the quality of the code decreased through the versions.

Table 6. Distribution of general issues between jEdit versions

Project	CODE_SMELL	BUG	VULNERABILITY
jEdit 5.5	97.14%	2.86%	-
jEdit 5.6	90.40%	9.59%	0.01%

Table 7. Distribution of Java issues between jEdit versions

Project	CODE_SMELL	BUG	VULNERABILITY
jEdit 5.5	20.79%	0.32%	-
jEdit 5.6	31.85%	0.85%	0.01%

5.2. **Freemind 1.0.1 and 1.1.0.** For Freemind, we extracted a total of 12653 issues, from which 12349 labeled as code smells, 302 labeled as bugs and 2 labeled as vulnerabilities. As opposed to jEdit, the number of Java issues comprises the majority of general issues, with a number of 12549, from which 12269 code smells, 278 bugs and 2 vulnerabilities. Data can be visualised in Tables 8 and 9.

From a percentages point of view, we can observe that the ratio is similar for both general issues and java issues, with an approximate 97% and 2% percent of code smells and bugs holding between releases 10 and 11. The improvement here can be observed from the number of actual issues between the two versions, with a clear decrease of both code smells by 25% and bugs

Table 8. General issues comparison between Freemind versions

Project	Code Smell	Bug	Vulnerability
Freemind 1.0.1	12349	302	2
Freemind 1.1.0	9633	249	2

Table 9. Java issues comparison between Freemind versions

Project	Code Smell	Bug	Vulnerability
Freemind 1.0.1	12269	278	2
Freemind 1.1.0	9547	225	2

by 20%. While explicit vulnerabilities did not change, their number are too few to be relevant in this context.

We can say that the quality of code improved by 20% between releases, as opposed to jEdit, where the severity increased.

Table 10. Distribution of general issues between Freemind versions

Project	Code Smell (%)	Bug (%)	Vulnerability (%)
Freemind 1.1.0	97.60	2.39	0.02
Freemind 1.0.1	97.46	2.52	0.02

Table 11. Distribution of Java issues between Freemind versions

Project	Code Smell (%)	Bug (%)	Vulnerability (%)
Freemind 1.0.1	96.97	2.20	0.02
Freemind 1.1.0	96.59	2.28	0.02

5.3. **TuxGuitar 1.5.2 and 1.5.3.** TuxGuitar analysis shows a total of 3296 total issues for version 1.5.2, with a number of 3012 code smells, 258 bugs and 26 vulnerabilities. For version 1.5.3, we registered a total of 2930 issues, from which 2746 code smells, 184 bugs and 18 vulnerabilities. We can already observe an improvement from v1.5.2 to v1.5.3, as all categories of issues had a clear decrease. Data is shown in Table 12. A main difference from the previous two inspections shows that, for TuxGuitar, our test produced only java issues, meaning that the TuxGuitar projects that we analysed did not include other types of resources such as .html and .xml files that could have been analysed by SonarQube in the way we ran the tool. Hence, we did not publish a second table as the data between general issues and Java only issues is not different.

Table 12. General issues comparison between TuxGuitar versions

Project	Code Smell	Bug	Vulnerability
TuxGuitar 1.5.2	3012	258	26
TuxGuitar 1.5.3	2746	184	18

The distribution percentage of issues between versions has a similar ratio, while showing a slight increase in code smells and a slight decrease in bugs. This shows an improvement in the overall code base and a decrease in severity, as the percentage of bug issues is smaller than code smells when reported to the total number of issues. The improvement is better than in Freemind's case, where, even though every category of findings improved, the percentage of bugs related to the total issues became higher in the newer version. Table 13 contains the distribution of TuxGuitar issues.

Table 13. Distribution of issues between TuxGuitar versions

Project	Code Smell (%)	Bug (%)	Vulnerability (%)
TuxGuitar 1.5.2	91.38	7.83	0.79
TuxGuitar 1.5.3	93.15	6.24	0.61

5.4. **Comparison between the three datasets.** All three datasets showed improvements in the number of issues found between versions. This indicates that continuous development and refactorings decreased the number of total issues for each project. Even though the total number of issues has shown improvement, the quality of the changes was different:

- jEdit introduced more bugs than before, more specifically an addition of 2122 SonarQube BUG rule violations
- Freemind kept a similar ratio between code smells and bugs, with bugs slightly taking more space in the newer version
- TuxGuitar had the best result, with a similar ratio between code smells and bugs, and bugs also decreasing from a distribution point of view

Relating the above results with the maintainability index computed from Section 4, we can conclude the following:

- Starting with jEdit as the first analyzed project, we concluded that the overall code quality decreased between releases, even though the total number of issues has improved. This is backed by the fact that the number of bugs introduced in a newer release were superior (from 2% to 9%) than the older release and the fact that bugs hold

a higher severity than code smells. The flow of development in this case seems to have brought down the overall quality of the project, asking the question of what might have happened in the development process. There may be a correlation between a big number of issues that the project has and the difficulty of maintaining and developing a complex application, hence the decrease in code smells and the increase in bugs in vulnerabilities. This can mark the subject of future research.
- With Freemind, the development process showed linear progress in the quality of the code. Both code smells and bugs issues have a similar percentage between releases, with an overall improvement of the project quality. Vulnerabilities were not taken into account due to their low number. This improvement may signify a better management in development processes and a greater attention to detail than the other projects when it comes to solving issues. The lower number of issues than jEdit may also pose a reason for this result, bringing into consideration the possible correlation from the previous point. Against all evidence, the maintainability index classified the project as "Bad", showing that the metric may not be generally applicable to empirical studies on refactorings.
- Finally, it can be observed that TuxGuitar's shift from version 1.5.2 to 1.5.3 not only exhibited a decrease in overall problems but also reflected an improvement in its maintainability index. The software obtained a "Satisfactory" rating in this area. The success of its refactoring endeavors is evident in the positive trajectory, which is characterized by a decrease in bugs and vulnerabilities. Simultaneously, the increase in code smells highlights the significance of ongoing emphasis on refining coding practices, in a way that both maintainability and overall code quality are maintained in future iterations.

6. Conclusions and future work

This study focuses on the analysis of software source code and its impact on software maintainability, considering factors such as cost and time allocation. The use of artificial intelligence has gained prominence in analyzing software problems. The study explores the relationship between software maintainability, technical debt issues, and code refactorings. The objective is to develop high-performing approaches for predicting software maintainability and the number of code refactorings based on technical debt issues.

To ensure accurate prediction of software maintainability and the number of required code refactorings, a comprehensive dataset was essential. This study focused on three open-source Java projects: jEdit, FreeMind, and TuxGuitar. Technical debt issues were obtained by performing static analysis on specific versions of these projects (jEdit 5.5, FreeMind 1.0.1, and TuxGuitar 1.5.2). Refactoring data was obtained by using the RefactoringMiner tool to compare different versions of each project. The technical debt data from the SonarQube tool was combined with the RefactoringMiner data and processed accordingly.

Two different approaches were considered. The first approach consolidated the technical debt issues to a single issue per software component, while also including the count of technical debt issues per component as an additional feature. Data augmentation techniques, such as Noise Injection, were applied to balance the dataset for increasing the generalization capacity of the models by simulating different variations in the data through the addition of random noise. Classical machine learning algorithms, including the Multi-layer Perceptron, Decision Trees, Random Forest, and Support Vector Machine, were employed as intelligent algorithms. The ExtraTrees algorithm yielded the best results in both the classification and regression tasks, achieving an accuracy of 0.92, F1-Score of 0.92 (classification), R-Squared of 0.92, and RMSE of 2.75 (regression).

The second approach did not modify the technical debt issues associated with each software component, allowing for the possibility of multiple entries for the same component in the dataset. A Recurrent Neural Network (RNN) was employed as the intelligent algorithm for this approach. However, the RNN model did not perform as well as the ExtraTrees algorithm in the first approach, achieving only an accuracy of 0.57 (classification) and R-Squared of 0.50 (regression).

The proposed methodology employs technical debt data to forecast the maintainability of software and the required number of code refactorings in three open-source Java projects. However, it is important to acknowledge a potential threat to the validity of the obtained outcomes due to the specific use of Java projects. Consequently, when applied to projects developed in different programming languages, the reliability of the results and the model's performance may be compromised.

Furthermore, certain aspects were not considered in the current approach that warrant exploration. These aspects include specific details pertaining to technical debt issues (e.g., message content) and refactorings (e.g., refactoring type and description). Integrating the message content of issues and the type or description of refactorings into the prediction model has the potential to enhance accuracy and provide more insightful information to end users. Additionally, the current implementation solely relies on technical debt data, and future enhancements could involve incorporating additional software metrics.

Such metrics offer valuable insights into the code's state and their inclusion could improve the predictive model.

Another area for investigation involves refining the performance of the second proposed approach, as it does not condense technical debt data to a single entry per software component. Similarly, in the first approach, reducing technical debt data to a single entry per component is achieved by calculating the mean of all values. However, it is essential to note that this may not be the most optimal reduction method. Therefore, this presents an additional area that requires further investigation and refinement.

The attained results represent a step towards building a strong predictive model for software maintainability and the necessary number of code refactoring. Additionally, these outcomes can be easily integrated into a web application, ensuring convenient access. This advancement has the potential to aid the software community in improving their assessment of software maintainability, thereby contributing to reduced resources needed for the maintenance phase.

References

[1] Akour, M., Alenezi, M., and Alsghaier, H. Software refactoring prediction using svm and optimization algorithms. *Processes 10*, 8 (2022).

[2] Arisholm, E., Briand, L. C., and Johannessen, E. B. An empirical study on the relationship between software maintainability and bug-proneness. In *2010 IEEE International Symposium on Software Metrics (METRICS)* (2010), IEEE.

[3] Biau, G., and Scornet, E. A random forest guided tour. *TEST 25* (2016), 197–227.

[4] Breiman, L. Classification and regression trees. In *Decision forests for computer vision and medical image analysis* (2017), Springer, pp. 19–38.

[5] CAST. 2018 software intelligence report. Tech. rep., CAST, 2018.

[6] Cortes, C., and Vapnik, V. Support-vector networks. *Machine Learning 20*, 3 (1995), 273–297.

[7] Drucker, H., Burges, C. J., Kaufman, L., Smola, A. J., and Vapnik, V. Support vector regression machines. *Advances in neural information processing systems 9* (1997), 155–161.

[8] Elmidaoui, S., Cheikhi, L., Idri, A., and Abran, A. Machine learning techniques for software maintainability prediction: Accuracy analysis. *Journal of Computer Science and Technology 35*, 5 (2020), 1147–1174.

[9] Ernst, N. A., and Eichmann, D. A. The future of software maintenance. *IEEE Software 16*, 1 (1999), 44–50.

[10] Geurts, P., Ernst, D., and Wehenkel, L. Extremely randomized trees. *Machine Learning 63*, 1 (2006), 3–42.

[11] Guo, G., Wang, H., Bell, D., Bi, Y., and Greer, K. Knn model-based approach in classification. In *On The Move to Meaningful Internet Systems 2003: CoopIS, DOA, and ODBASE* (Berlin, Heidelberg, 2003), R. Meersman, Z. Tari, and D. C. Schmidt, Eds., Springer Berlin Heidelberg, pp. 986–996.

[12] Hegedűs, P., Kádár, I., Ferenc, R., and Gyimóthy, T. Empirical evaluation of software maintainability based on a manually validated refactoring dataset. *Information and Software Technology 95* (2018), 313–327.

[13] Jang, J.-S., Sun, C.-T., and Mizutani, E. *Neuro-fuzzy and soft computing: a computational approach to learning and machine intelligence.* Prentice Hall, 1997.

[14] Kaur, A., and Kaur, K. Statistical comparison of modelling methods for software maintainability prediction. *International Journal of Software Engineering and Knowledge Engineering 23*, 6 (2013), 743–774.

[15] Ke, G., Meng, Q., Finley, T., Wang, T., Chen, W., Ma, W., Ye, Q., and Liu, T.-Y. Lightgbm: A highly efficient gradient boosting decision tree. In *Advances in Neural Information Processing Systems* (2017), I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds., vol. 30, Curran Associates, Inc.

[16] Marinescu, R. An empirical study of the relationship between code smells and refactoring. *Empirical Software Engineering 9*, 4 (2004), 429–462.

[17] Molnar, A.-J. Collection of technical debt issues in freemind, jedit and tuxguitar open source software.

[18] Molnar, A.-J., and Motogna, S. Long-term evaluation of technical debt in open-source software. In *Proceedings of the 14th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)* (New York, NY, USA, 2020), ESEM '20, Association for Computing Machinery.

[19] Molnar, A.-J., and Motogna, S. A study of maintainability in evolving open-source software. In *Evaluation of Novel Approaches to Software Engineering* (Cham, 2021), R. Ali, H. Kaindl, and L. A. Maciaszek, Eds., Springer International Publishing, pp. 261–282.

[20] Montgomery, D. C., Peck, E. A., and Vining, G. G. *Introduction to linear regression analysis.* John Wiley & Sons, 2012.

[21] NIST. The economic impacts of inadequate infrastructure for software testing. Technical Report NISTIR 6859, National Institute of Standards and Technology, 2002.

[22] Oman, P., and Hagemeister, J. Metrics for assessing a software system's maintainability. In *Proceedings Conference on Software Maintenance 1992* (Nov 1992), pp. 337–344.

[23] Pearl, J. Probabilistic reasoning in intelligent systems: Networks of plausible inference. *Morgan Kaufmann* (1988).

[24] Rumelhart, D. E., Hinton, G. E., and Williams, R. J. Learning representations by back-propagating errors. *Nature 323*, 6088 (1986), 533–536.

[25] Taud, H., and Mas, J. *Multilayer Perceptron (MLP).* Springer International Publishing, Cham, 2018, pp. 451–455.

[26] van Koten, C., and Gray, A. R. An application of bayesian network for predicting object-oriented software maintainability. *Information and Software Technology 48*, 1 (2006), 59–67.

[27] Wahler, M., Drofenik, U., and Snipes, W. Improving code maintainability: A case study on the impact of refactoring. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)* (2016), pp. 493–501.

Departyment of Computer Science, Babeş-Bolyai University, Cluj-Napoca, Romania

Email address: liviu.berciu@ubbcluj.ro, vasilica.moldovan@stud.ubbcluj.ro