BABEŞ BOLYAI UNIVERSITY, CLUJ NAPOCA, ROMÂNIA
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE

Forecasting The Need for Software Refactorings based on Technical Debt Issues

– ITSG report –

Moldovan Vasilica, Software Engineering PhD Student

2022-2023

Abstract

As the software engineering sector continues its expansion, there is a growing emphasis on its quality factors. Consequently, software refactoring processes are once again gaining attention. The primary goal of code refactoring is to enhance readability, simplify complexity, eliminate architectural flaws, and ultimately improve code maintainability. Simultaneously, these efforts aim to make the code more flexible and easier to extend and work with. The benefits of effective software refactoring are numerous.

However, a critical question arises: when should these refactorings be performed? Numerous studies have investigated the optimal timing for software refactorings, leading to the development of tools in this domain. Most findings suggest that the decision to refactor often rests with the programmer, as specific tasks or time slots for software refactoring are typically not allocated within the industry. Consequently, software refactoring relies heavily on the engineer's experience. The limitations of existing tools for predicting software refactorings often stem from a lack of adaptability to modern language features, infrequent updates, and inadequate learning from diverse and current datasets. These factors contribute to performance issues, potentially rendering some tools less effective and misaligned with contemporary software development practices.

In this research, a machine learning approach was explored to forecast the necessity of software refactoring and, if required, to identify its type based on technical debt issues identified in static analysis. The developed prediction model has been integrated into a web application directly linked to an active SonarQube session, simplifying access to the required input data. This web application caters to software engineers, students, and other practitioners contemplating code refactoring.

Various methodologies were investigated, including consolidating input data into a single instance per software component or retaining multiple instances per component to capture crucial information comprehensively. Diverse machine learning algorithms were employed to predict the software refactoring type, if needed. Notable algorithms in use encompass ExtraTreeClassifier, DecisionTreeClassifier, RandomForestClassifier, and RNN classifiers.

The input dataset comprised technical debt information from three open-source Java projects, and the output data involved 28 possible classes: 27 refactoring types and a class indicating that no refactoring is presently required. In order to increase the performance of the model by balancing the dataset, several data augmentation techniques were applied. The results achieved were promising, with the highest-performing model achieving an accuracy of 0.96.

Contents

List of Figures

Chapter 1

Introduction

1.1 What? Why? How?

The aim of this paper is to realize a web application which predicts the type of the software refactoring process needed by each software components, any refactoring is needed based on the technical debt issues found in that project. The output classes are 28, namely the 27 refactoring types defined by Fowler [4] and one output class corresponding to the scenario in which no refactoring process is needed at the moment. The 27 refactoring types considered are: Add Attribute Modifier, Add Method Annotation, Replace Anonymous With Lambda, Change Variable Type, Replace Loop With Pipeline, Add Attribute Annotation, Move Attribute, Merge Catch, Change Attribute Access Modifier, Extract Method, Remove Method Modifier, Change Parameter Type, Rename Method, Split Conditional, Change Return Type, Replace Variable With Attribute, Add Parameter Annotation, Move And Rename Class, Inline Method, Rename Parameter, Change Attribute Type, Push Down Attribute, Add Parameter, Change Method Access Modifier, Rename Attribute, Rename Variable, and Rename Class.

The recognition of the optimal timing for executing a software refactoring process has gained increasing significance. This stems from the substantial growth observed in the software development domain over recent decades, with this growth exhibiting an ongoing upward trajectory. Given that software refactoring aims to enhance code readability, simplify code structure, and eliminate potential design flaws, these objectives directly result in heightened software maintainability. Consequently, this translates to reduced resource requirements during the maintenance phase of each project, an imperative phase with costs comparable to the development phase, as indicated by statistical analyses [3].

Furthermore, a clear and readable codebase is more amenable to modifications for incorporating new features, fostering increased flexibility and extensibility. The reduced resource demand during

maintenance also frees up resources that can be reallocated to the development phase, thereby promoting the advancement of new software initiatives.

An additional pivotal consideration emphasized in [4] underscores the need for cautious decision-making when embarking on a software refactoring endeavor. Not all types of refactorings are universally applicable, and certain types may pose greater implementation challenges than others. Therefore, deciding the opportune moment for software refactoring without unnecessarily consuming time or complicating the code holds high importance. Despite the availability of multiple tools suggesting potential refactorings, these tools see limited adoption in the industry due to their tendency to yield numerous false positives. The limitations of existing tools for predicting software refactorings often arise from an inability to adapt to modern language features, infrequent updates, and insufficient learning from diverse and contemporary datasets. These factors contribute to performance issues, potentially diminishing the effectiveness of some tools and misaligning them with current software development practices.

For the machine learning component of the system, two distinct categories of models were employed. The initial category containes classifier models sourced from the LazyPredict library, while the second category involves an RNN model constructed using the Keras library. The ultimate determination regarding which model will be incorporated into the application will be made subsequent to conducting various experiments and scrutinizing their outcomes. Two different methodologies were adopted based on the model category, each with respect to the input data:

- For the models from LazyPredict, all technical debt issues were compressed into only one entry per each software component. Several reduction methods were considered among which: average mean, sum, median value, and frequency.
- For the RNN model, technical debt issues were not compressed into only one entry per each software component. This decision made because this way useful information might be kept, information that would have been otherwise lost in the reduction process.

Since this problem is one of great interest, it has been addressed previously. There are many studies done in this area, and various approaches have been chosen. This paper chose a machine learning approach for predicting the type of software refactoring needed by each software component, approach that uses an intelligent model, the classification being done technical debt issues data. In end, this model is integrated into an Flask web application, in order to be more practical for all users.

1.2 Paper structure and original contribution(s)

The research presented in this paper advances the theory, design, and implementation of several particular models.

The main contribution of this report is to present an intelligent algorithm for solving the problem of predicting the type of the needed software refactoring process for each software component, if a refactoring is needed.

The second contribution of this report consists of building an intuitive and easy-to-use and software application. This aim was to build an algorithm that will help all users (more or less experienced) to identify 27 types of refactoring opportunities and the possibility that no refactoring is needed at the moment, by analyzing the technical debt information available after running the static analysis process. A web application that can assist the practitioners to more accurately identify refactoring opportunities will be truly helpful. This is due to the fact that most of the time, the decision to perform or not certain refactorings is based on programmer's experience and intuition, since there are are no rigorous recipes for when and where to perform all types of refactorings. Since performing refactorings can also diminish the quality of the software if those refactorings are performed when they are not needed, the practitioners need to carefully consider all the implications before performing a refactoring process upon their code.

The present work contains [4], [3], [6], [5], [1] bibliographical references and is structured in five chapters as follows.

The first chapter is a short introduction into what the problem is more exactly, why it is important to be addressed and how it is planned to solve it. There has been made a short presentation on the importance of software refactoring processes in the software industry, the fact that software refactorings represent a very powerful technique of improving the maintainability of the software systems, reducing the costs associated to the maintenance phase. Software refactorings have as purpose to make the code simpler, more clear to the engineer and to increase the flexibility and extensibility of the code.

The second chapter describes is more details the scientific problems this paper aims to solve. In 2.1 more data about the problem of correctly using software refactoring processes is given.

In chapter 3, it is presented the state of the art of predicting software refactoring opportunities and automated tools built in this purpose, giving more details about the best performing approaches and the results that were obtained.

The fourth chapter details the proposed approach, giving more details on the chosen classification technique and how everything was combined to build the web application, which is the final aim.

In chapter 5, it is described the result of all the research done in this paper. In section 5.1 the chosen machine learning model is discussed in more details. After that, it follows a presentation of the dataset used for training and testing the model, in section 5.2. In section 5.3 the results from all performed experiments are presented, while in section 5.4 an interpretation of those results is shown.

In chapter 6, an analysis of the strengths, weaknesses, opportunities and threats is discussed, considering the size and the time performance of the web application, the performance of the model and the ethics implied in the entire process and in using the final web application by each type of users.

Finally, in chapter 7, the conclusions are drawn and possible future directions of work are shown.

Chapter 2

Scientific Problem

2.1 Problem definition

Software maintainability is one of the most important quality factors of the software systems. Being defined as "The ease with which a software system or component can be modified to correct faults, improve performance or other attributes, or adapt to a changed environment" [6], maintainability has proven that it is absolutely essential for the long-term success of any software product, directly influencing its quality.

There are several studies that reflect the negative effects a software product with low maintainability brings. A 2018 study performed by CAST revealed that poor software maintainability costs the US economy over $2.8 trillion annually [**?**]. Also, according to the National Institute of Standards and Technology (NIST), software maintenance activities account for up to 90% of the total cost of software development [**?**].

There is a strong connection between the software systems' detected issues, its maintainability and the refactorings performed on that software. 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. Technical debt is defined as the cost of opting for a quick and simple solution over a more resilient and scalable one with the awareness that the quick solution would need more effort and maintenance in the future. It can manifest in many forms such as code smells, bugs or vulnerabilities, having an inversely proportional relationship with software maintainability.

There is a strong connection between the software systems' detected issues, its maintainability and the refactorings performed on that software. Code refactoring is the process of restructuring existing code without altering its functionality externally. 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 over-complicate 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.

An automatic way to detect software refactoring opportunities based on technical debt issues, assisting the software engineers when deciding if certain refactoring processes should be performed would be of much use, since the decision of performing software refactorings needs to be taken after careful consideration due to the many implication such a process has.

The main challenges encountered here were distributed into different categories, being the following:

- Firstly, because there were no previously studies of refactoring opportunities detection based on static analysis, there was no dataset suitable for that. In order to obtain the data needed for training the classification/prediction model two different of datasets were combined:
  - A dataset containing the report obtained after performing static analysis. On this dataset a few other operations were performed in order to extract only the needed data.
  - A dataset containing all the refactorings done between two versions of the same software product.
- Another interesting aspect to explore was how to increase the performance of the model, in order to have a more reliable application.
- Furthermore, another challenge was represented by the evaluation of the final results. Since the approach is different than the others in the literature, it doesn't have a direct correspondent to which it can be compared. Different metrics were studied in order to analyze better the results, and the model was tested on several other open source projects.
- Another challenging topic was represented by the final integration of the classification model into an application, being important that the application should be easy to use and should require the users to do as few preparation on their data as possible.

Building an application that takes the report resulted after performing a static analysis of the source code and accurately detects the refactoring opportunities of that code would be truly helpful for all parties involved in the development and maintenance process of any software product. It would

reduce the costs of the maintenance phase, and would allow the developers to concentrate their time and resources more to improve and add new capabilities to their software, being able to create and build software products that might have a more important role on a large scale, in different segments of expertise. Also, a higher maintainability would imply a smaller number of issues, increasing the security of software products.

Chapter 3

Related work

Software systems have become more and more popular in the last decades, and now people cannot imagine life without the help of software. This represents a challenge due to the increasing demand of new and innovative software systems, all while preserving and updating existing systems. The correctness and the efficiency of any current software system is a must, as they are also used in many safety-critical domains. Being one of the most important quality factors, software maintainability helps with ensuring long-term sustainability and reducing technical debt.

The refactoring process is one of high importance due to its capability of improving code quality, enhancing maintainability, and promoting collaboration among developers. Refactoring helps eliminate code smells, reduce technical debt, and optimize code performance, leading to more robust and scalable software systems. All these imply a strong and direct relationship between the number of performed refactorings and the maintainability of the source code. As a consequence of this relationship, many research studies have been performed aiming to better analyze the relationship between refactoring and software maintainability, and also aiming to predict the need of refactorings.

The paper [5] introduces an improved dataset of validated refactoring data for open-source systems. It shows that refactoring is commonly applied to entities with low maintainability, indicating that developers actively address deteriorated code. Metrics related to size, complexity, and coupling are significantly higher in refactored elements, suggesting that developers target these areas for improvement. However, clone-related metrics are found to be less significant in the analysis. The results offer valuable insights into refactoring practices and can inform the development of tools aligned with developers' habits. It should be noted that the improved dataset covers only a fraction of the base dataset due to the extensive validation process involved. The findings contribute to a deeper understanding of refactoring and pave the way for further research in this field.

The paper [1] presents a study focused on predicting software refactoring using Support Vector

Machine (SVM) and optimization algorithms. It addresses the correlation between code coverage and test suite effectiveness in an evolutionary manner. The authors explore the use of SVM with genetic and whale algorithms to predict refactoring at the class level. The study uses a dataset from open-source software systems and achieves promising accuracy rates ranging from 84% to 93%, with improved performance when merging SVM with the optimization algorithms. The proposed approach outperforms four other well-known machine learning algorithms in terms of prediction performance. The findings demonstrate the effectiveness of SVM and optimization algorithms for software refactoring prediction, highlighting their potential value in enhancing software quality.

The research paper [9] presents a class-based approach for predicting potential refactoring candidates. By employing a specific set of static metrics and a weighted ranking technique, a tool was developed to forecast a prioritized list of classes requiring refactoring. A comparative study was conducted to assess the tool's performance in identifying design issues that adversely impact class maintainability, in comparison to human reviewers. The findings of the study demonstrate the significant assistance provided by the refactoring decision support tool to software teams. The results obtained from the tool, which primarily focuses on size and complexity, exhibited a certain level of agreement with the evaluations made by human reviewers. This small-scale study provides initial support for the idea that complexity and size are significant factors that contribute to the difficulty programmers face in understanding legacy code and can be utilized to predict classes that require refactoring. The study demonstrates that an automated approach can establish a prioritized list that aids in cost-effective planning for refactoring. Furthermore, it was observed that while reviewers generally agreed on which classes should be refactored first, they often looked for different types of design issues. Around 28% of the identified code issues by the reviewers were not covered by the provided list of predefined code smells.

Another study worth mentioning is represented by [2], which performed a systematic literature review of automatic software refactoring. Different papers addressed model level refactoring and code level refactoring, while some of them converted source code into a class diagram and further used genetic algorithms to analyze it. Different tools have been created, most of them as plugins in Eclipse, but their usage has not been very spread. Most of the tools either perform refactorings automatically or detect already performed refactoring processes. One important aspect that needs to be mentioned is that all those studies perform their analyses on software metrics and not on technical debt issues data. For automatic refactoring, three mechanisms were used: search-based, machine learning and cloning, with the search-based methods being the most used, aprox. 65%. To detect refactoring opportunities, models using predefined rules or machine learning approaches were employed.

Chapter 4

Investigated approach

The aim of this project is to create an intuitive web application which successfully detects refactoring opportunities based on technical debt issues as accurate as possible.

Firstly, when all technical debt issues were compressed only one entry per each software component, the detection of software refactoring opportunities was done using a classifier from the lazypredict.Supervised.LazyClassifier library from python. Different reduction methods were investigated, such as: average mean, the total sum, and median value. For this models such as ExtraTrees, DecisionTrees, RandomForest, SVC, LGBM, and others. Those classifiers were all trained with their default parameters. Then a second approach was tried, by using a custom RNN model in the scenario in which the technical debt issues were not compressed into only one entry per each software component. The model had the following componence:

- An Embedding layer.
- Following, a LSTM, long-short term memory, layer was added.
- A Dropout layer followed.
- Another LSTM layer followed.
- Following, a new Dropout was added.
- A Dense layer was added further.
- Next, a new Dropout layer, identical to the previous two was added.
- The last layer was represented by a Dense layer.

After training the model, the one that obtained the best performance was exported as a .sav file and integrated into a Flask web application. This application can take as input either a .xlsx file representing the report of the static analysis or it can connect to SonarQube and retrieve the needed data for the specified project. If the chosen input type is the one uploading a file, then this file needs to contain a few specific columns in order to make an accurate prediction, namely: the component name, the severity of each issue, the debt of each issue and the type of each issue. The file might contain other information, but those four columns are mandatory for the prediction to work. For the SonarQube input option, the only requirement is to have an open SonarQube session in that specific moment. The application will pre-process the given inputs and extract the needed information for the prediction phase.

After uploading the input to the application, the user can see different data visualization based the uploaded input, for both .xlsx file and SonarQube input types. The statistics available are about the number of technical debt issues that were detected in each software component, their severity and their type, giving the user the possibility to also export these charts.

The results are given in the form of the table, table that states for each software component the type of the refactoring needed or "No refactoring is needed" in the scenario in which the code does not need a specific refactoring at the moment. The user has the possibility to download the resulted report as a .xlsx file, and also to visualize the statistics regarding the types of the predicted refactorings, while for those, also having the possibility to download them.

The integration of the intelligent model into the web application was straight-forward, the only weakness of this approach being represented by the substantial amount of time needed for the prediction, this happening when the project that needs to be analyzed is one of large proportions. This is due to the fact that before actually running the intelligent algorithm, the input data needs to be processed and prepare for the algorithm, and this increases the time needed by the application to provide the requested results.

The flow described above is presented in 4.1.

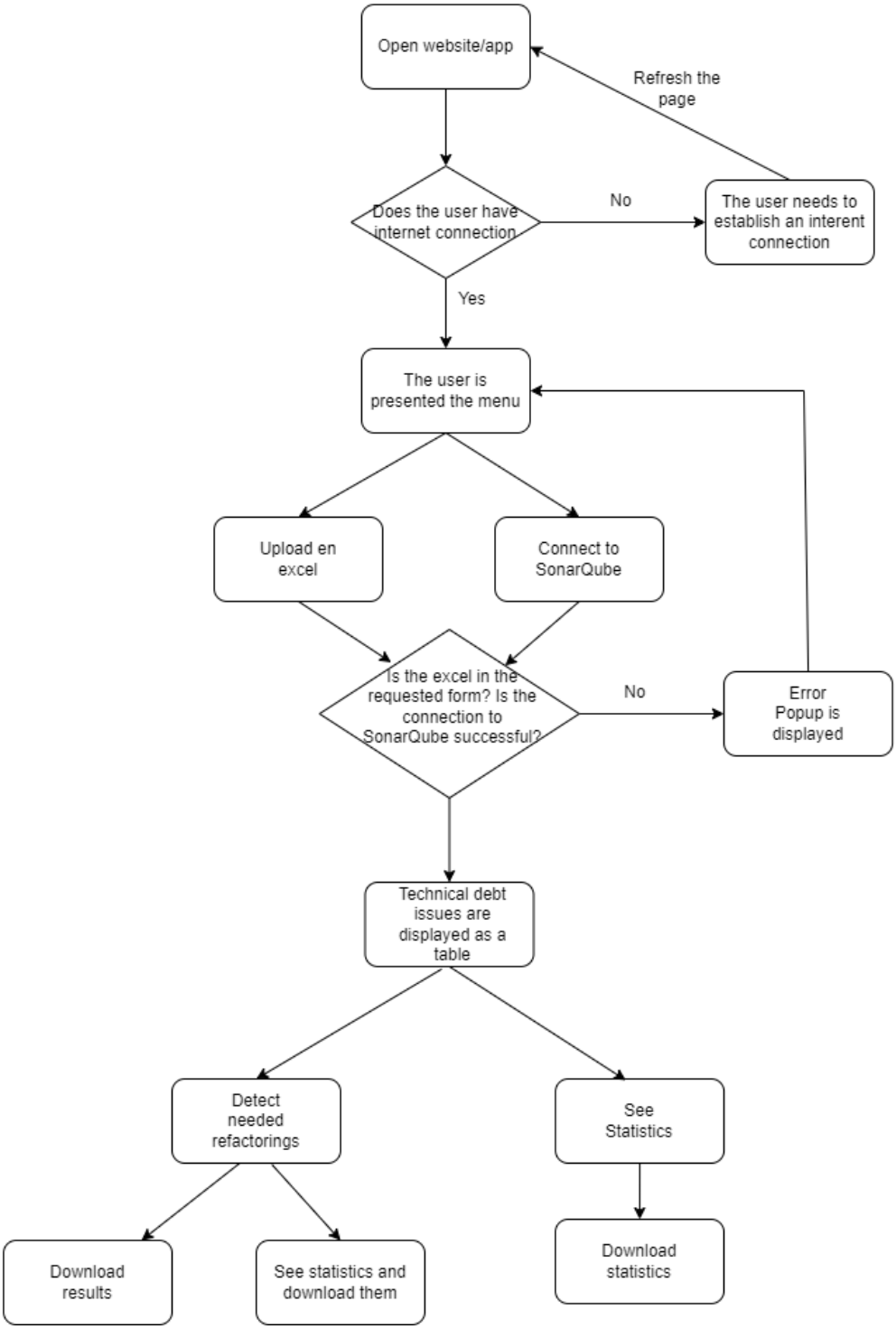


Figure 4.1: Description of the flow.

Chapter 5

Application (numerical validation)

Building an accurate system that detects software refactoring opportunities and integrating it into a web application was the purpose of this project. Several experiments were performed for both determining the best model to predict the software refactoring types and the best way to integrate this model into a Flask web application, considering all aspects which might interest the final users. In the following, the methodology used in this approach is presented.

5.1 Methodology

For detecting the software refactoring opportunities, there were two options for the model:

- Using the first dataset, the one containing only one entry per each software component, several models were trained obtaining a good performance. LazyClassifier from lazypredict.Supervised library was used, and the following models were studied: Extra Tree, LGBM, Random Forest, K-Neighbors, Decision Trees and SVC. On these models, using cross val score from sklearn.model selection, cross validation was performed with 5 folds, and the scoring metric being f1 macro The data was evenly split as following: 0.8 for training and 0.2 for testing.
- The second approach, the one containing multiple entries per component, each entry corresponding to a technical debt issue detected on that component, was used for a recurrent neural network. The model had the following componence:
  - An Embedding layer was added having the length equal to the length of the training data.
  - Following, a LSTM, long-short term memory, layer was added having as activation functions *Relu* and *sigmoid*.

- A Dropout layer followed in order to reduce the probability of overfitting, with a dropout rate of 0.2.
- Another LSTM layer followed, having the same activation functions as the previous LSTM layer.
- Following, a new Dropout layer with a dropout rate of 0.2 was added.
- A Dense layer with its activation function being *Relu* or *Sigmoid* was added further.
- Next, a new Dropout layer, identical to the previous two was added. The dropout layer serves as a form of ensemble learning, where various subsets of neurons are trained on various mini-batches of data. It also effectively introduces noise. This makes the network more resistant to overfitting by preventing it from depending excessively on particular features or combinations of features.
- The last layer was represented by a Dense layer, which had 41 (the number of refactoring types classes) output channels for classification. The activation function used was $Softmax$.

In order to assess the performance of the models, several types of evaluations techniques were considered. The initial measuring criterion was ACC (accuracy), which measures the classification's accuracy and has the formula $ACC = \frac{Number\ of\ correct\ predictions}{Total\ number\ of\ predictions} = \frac{TP+TN}{TP+TN+FP+FN}$, where TP is the number of true positives, FN is the number of false negative, FP is the number of false positive and TN is the number of true negative. Recall (or sensitivity), R, which is the proportion of retrieved instances to all relevant instances, was another evaluation criterion that was taken into consideration. The formula for the recall is as follows: $R = \frac{TP}{TP+FN}$, where TP and FN stand for, as stated above, the number of true positives and false negatives, respectively. In order to complement the recall, the precision metric was also computed. Precision measures the proportion of true positives among all predicted positives, having the formula: $P = \frac{TP}{TP+FP}$. Having the precision and recall, a natural direction was to compute the F1 score, since it provides a single value that takes into account both recall and precision. F1 score, defined as $F1Score = \frac{2*P*R}{P+R}$, is very useful for minimizing the appearances of false positives and false negatives is wanted, and also provides a good insight for the situation in which the dataset is imbalanced, situation in which accuracy is not enough in order to evaluate the performance of the model.

5.2 Data

In this study, three open-source Java projects were analyzed, namely: jEdit, FreeMind, and TuxGuitar. For each of these three projects, two types of data were taken into consideration: the technical

debt issues found on jEdit version 5.5, FreeMind version 1.0.1 and TuxGuitar version 1.5.2, and the refactorings performed between these versions of each project and the following versions: jEdit 5.6, FreeMind 1.1.0 and TuxGuitar 1.5.3.

The data regarding the technical debt issues found on these projects was taken from [7] this data being obtained after performing static analysis upon those three projects using the SonarQube tool. This dataset includes all the information about the technical debt issues that was provided at the end of the static analysis, for each issue specifying its project, its key, the rule that was used for finding it, its severity, the component in which it is located, its resolution, its status, its message, its debt, its creation, update and close data, its type and other useful tags. In this study however, not all attributes of the technical debt issues were considered. The attributes that were further investigated were the severity, the debt and the type of each issue, mapping those three characteristics to the component and the project in which the issue is located. This decision was taken because some of the attributes, such as the key, didn't necessarily bring relevant information to the classification/regression model. Additionally, some of the attributes required a more thorough and complicated examination and pre-processing than others, so those were left for future work. Along with the severity, debt and type of each issue, the number of issues per component was also computed and further used in the prediction process. There were detected 54617 issues in all three projects, while for each project, there was the following distribution of issues: jEdit 5.5 had 26023 issues, FreeMind 1.0.1 had 16962 issues, and TuxGuitar 1.5.2 had 11642 issues. By their type, the issues was classified into three categories:

- Code Smells: There were detected 45453 code smells, from which 20542 in the jEdit project, 14712 in the FreeMind project and 10199 in the TuxGuitar project. Code smells are defined as symptoms in the source code that may point to the presence of a problem, frequently a violation of design principles or coding standards, and the presence of a large number of them might indicate the need of a refactoring.

- Vulnerabilities: There were detected 1743 vulnerabilities, having the following distribution: 557 were found in jEdit project, 398 in the FreeMind project, and 788 in the TuxGuitar project. A rise in code vulnerabilities indicates the necessity for a refactoring since they signify software flaws that attackers could use to damage a system's security.

- Bugs: There were detected 7421 bugs, from which 4924 were found in the jEdit project, 1842 in the FreeMind project and 655 in the TuxGuitar project. An increase in code bugs indicates the need for refactoring since they signify mistakes or weaknesses in software that make it perform improperly or give inaccurate results.

In the matter of the severity, the issues were split into five categories, namely:

- Blocker: Overall, there were 376 blocker issues from which 145 were found in the jEdit project, 122 were found in the FreeMind project and 109 were found in TuxGuitar. Examples of blocker issues include unhandled exceptions, the use of deprecated classes or methods and others.
- Critical: There were detected 5537 critical issues, having the following distribution: 2122 issues were found in jEdit, 2508 in the FreeMind project and 907 in the TuxGuitar project. Some of those issues can be unsafe deserializations, unvalidated input, a too high cognitive complexity of a method and others.
- Info: There were found 1400 info issues, from which 350 were found in the jEdit project, 217 in the FreeMind project and 833 in TuxGuitar. The info issues can refer to removing the ToDos from the code, assignment of a value to a variable that is never used, unused private methods and so on.
- Major: There were detected 29644 from which 16157 critical issues were detected in the jEdit project, 7514 in the FreeMind project and 5973 in the TuxGuitar project. Examples of major issues are duplicated code, different magic numbers or hard-coded values in the code, unused import statements, regular expressions that are too complicated and may cause performance problems or wrong matches or others.
- Minor: There were found 17660 such issues, from which 7249 were found in the jEdit project, 6591 were found in the FreeMind project and 3820 were found in the TuxGuitar project. Some of the minor issues are represented by redundant casts, unused method parameters, empty statements and others.

For the final form of the technical debt issues data, two directions were approached:

- For each project, the technical debt issues were compressed to only one instance per component. For computing the final values of the severity and type of each software component, those two attributes were mapped to numerical values in the following way: $Severity = \{ INFO : 1, MINOR : 2, MAJOR : 3, CRITICAL : 4, BLOCKER : 5 \}$ and $Type = \{ CODE\_SMELL : 1, BUG : 2, VULNERABILITY : 3 \}$. After mapping the values of severity and type, either their mean, total sum or median value, was computed. For the debt, the values of N/A were not considered, they also being removed from the total number of issues per component. After performing these operations, each software component had only one

instance associated to it. After reducing all entries to only one per each software component, data was also normalized in order to improve convergence, prevents scale-related issues, and enhances robustness to outliers, ensuring fair feature contributions and compatibility with distance metrics.

- This other option didn't compress the issues instances to only one per class, but only removed the issues that had a N/A debt associated to them, as it was difficult for the model to interpret it, while also decreasing the number of issues per component that was previously computed.

The first option was built in order to comply with the input type of some well-known machine learning algorithms(Decision Trees Algorithms, SVM, KNeighbors, and others) that have obtained good results on both regression and classification problems on continuous data. The second option was build to comply with the requirements of recurrent neural network.

The data referring to the refactorings that were performed between the two considered versions of each project was obtained using the RefactoringMiner tool [8]. This tool needs two different versions of the same project, compares them and provides a report containing all the refactorings performed between these two versions of the project. The refactoring information provided is the following: for each refactor, the tool provides its type, a short description of what that refactor actually did and the component in which the refactor process was performed. All these attributes represent very useful information about the state of the code, giving insights and suggestions on its maintainability.

In order to construct the final dataset, the technical debt issues data and the refactorings data needed to be combined. The common element of those two datasets was represented by the software components, meaning that to the first datasets, a new column was added, column representing the type of software refactoring performed upon each specific component. The decision on which refactoring type needs to be associated to a software component was taken as described in the following:

- Firstly, each software component had associated a list of software refactorings performed upon it, as given from RefactoringMiner.

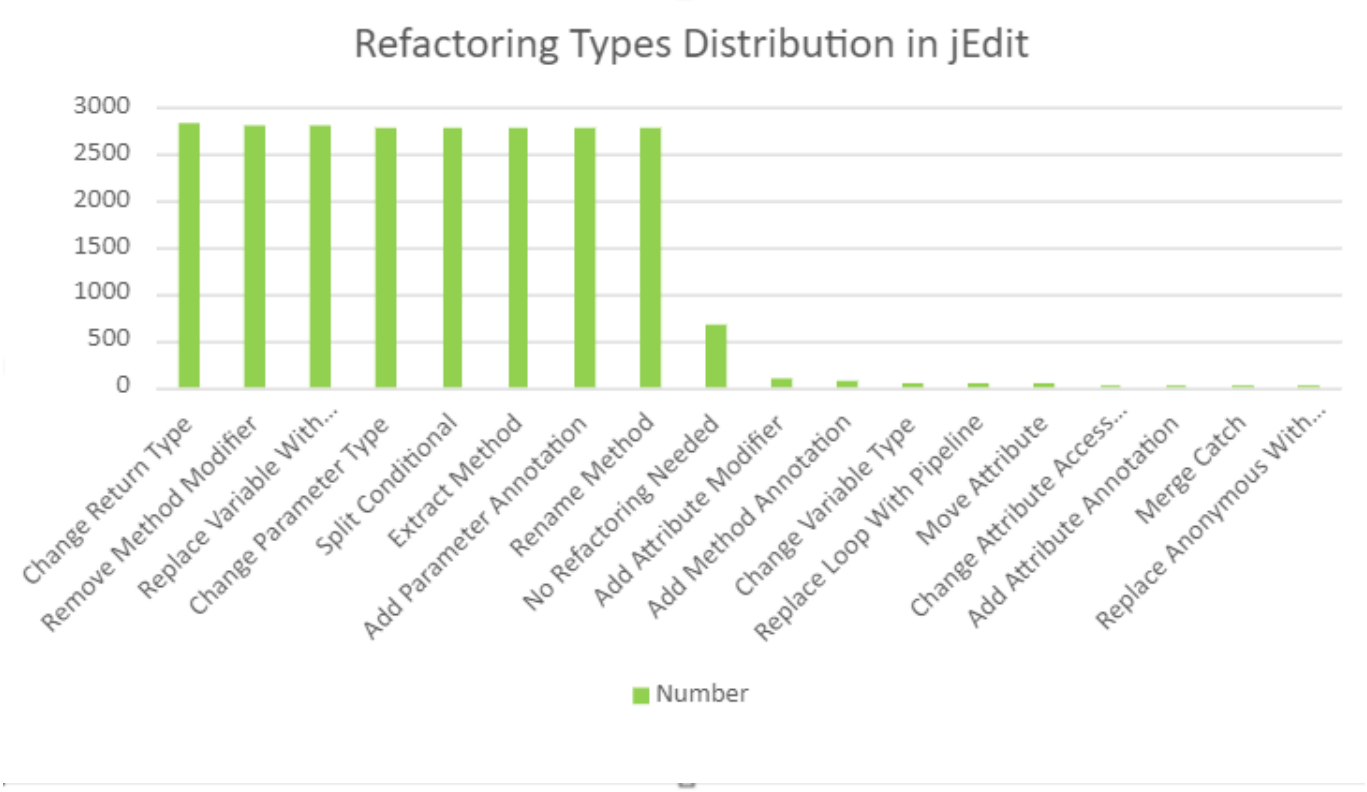
- Then, in order to compress this list into only one refactoring type, the most common refactoring from the list was kept.

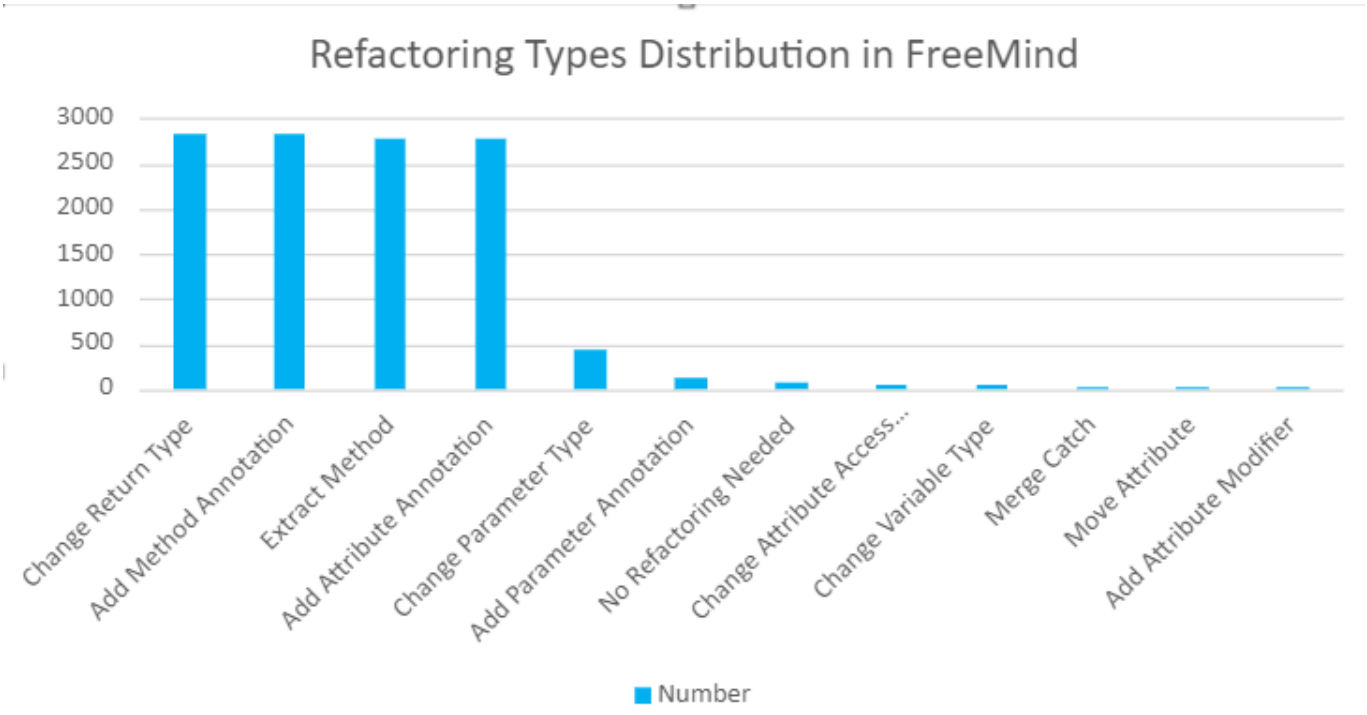
- If there were multiple refactoring types with the same maximum frequency, the one that was the least common overall was kept, in order to balance the dataset.

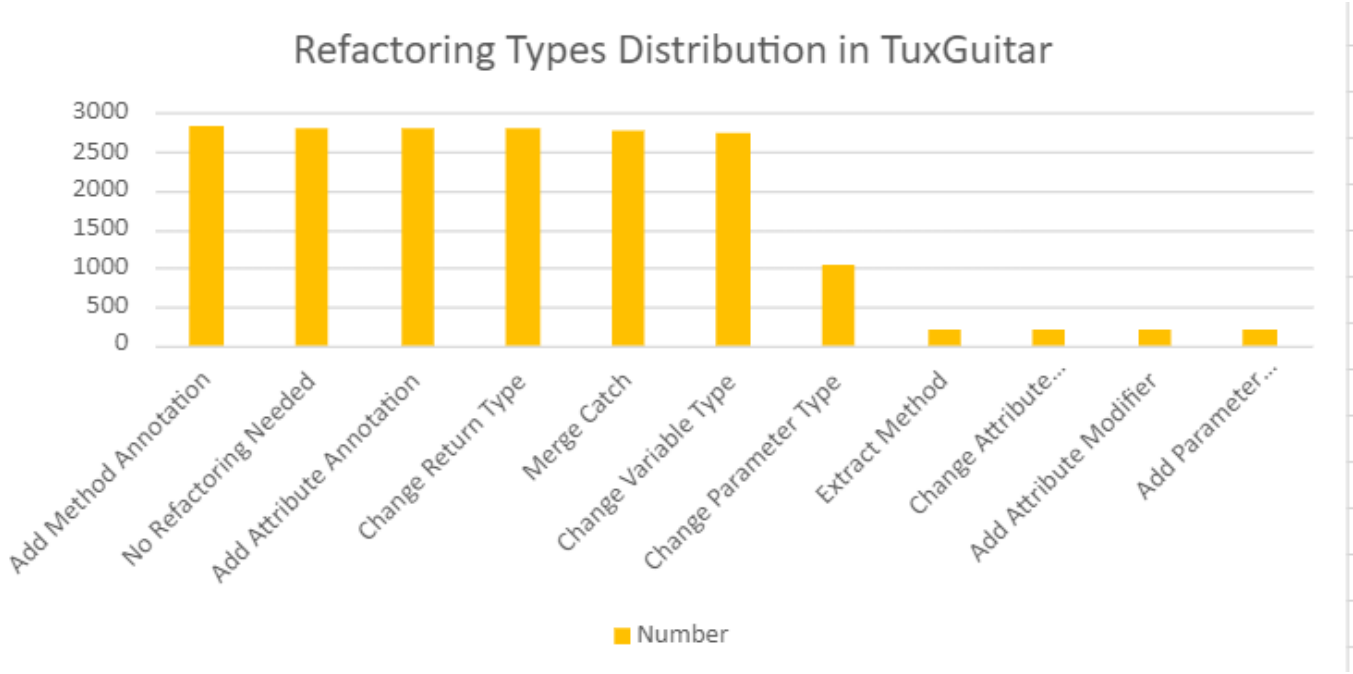
Hence, the final data that was analyzed and feed to the intelligent algorithm contained the following information: for each software component, the technical debt issues were presented by their severity,

debt, type and the type of refactoring performed on that software component.

Because both the versions of the dataset were quite imbalanced, data augmentation techniques were considered to improve the performance of the models. The SMOTE technique and Random Under-sampling were used to balance the class distribution in a dataset. SMOTE is applied to oversample the minority classes, and Random Under-sampling is applied to undersample the majority classes.







5.3 Results

In the following, we present the obtained results after performing several experiments employing the models from the LazyPredict library, which use the first option of the dataset, the one where all entries are compressed to only one per each software component. Those results are presented in 5.1, 5.2, 5.3, 5.4, 5.5, 5.6, 5.7, 5.8 and 5.9. For the second approach, the one using the dataset un-compressed only one entry per each software component and the RNN model built as described in 4, another row was added in those tables.

Model	Accuracy	Recall	Precision	F1-Score
RandomForestClassifier	**0.93**	**0.92**	**0.91**	**0.92**
ExtraTreesClassifier	0.91	0.90	0.90	0.90
KNeighborsClassifier	0.89	0.88	0.87	0.88
ExtraTreeClassifier	0.88	0.89	0.87	0.88
DecisionTreeClassifier	0.88	0.84	0.82	0.83
SVC	0.86	0.87	0.85	0.86
LogisticRegression	0.81	0.77	0.80	0.79
LinearSVC	0.79	0.75	0.75	0.75
Perceptron	0.69	0.67	0.68	0.68
RNN	0.65	0.65	0.65	0.65

Table 5.1: Results obtained for Lazy Classifier models after running the algorithm on jEdit data using **average mean** as a reduction method

The best results were obtained using the RandomForestClassifier model with the sum reduction

Model	Accuracy	Recall	Precision	F1-Score
RandomForestClassifier	**0.95**	**0.93**	**0.93**	**0.93**
ExtraTreesClassifier	0.92	0.91	0.91	0.91
KNeighborsClassifier	0.90	0.89	0.88	0.89
ExtraTreeClassifier	0.89	0.90	0.88	0.89
DecisionTreeClassifier	0.89	0.85	0.83	0.84
SVC	0.87	0.88	0.86	0.87
LogisticRegression	0.82	0.78	0.81	0.80
LinearSVC	0.80	0.76	0.76	0.76
Perceptron	0.70	0.68	0.69	0.69
RNN	0.66	0.65	0.65	0.65

Table 5.2: Results obtained for Lazy Classifier models after running the algorithm on jEdit data using **sum** as a reduction method

Model	Accuracy	Recall	Precision	F1-Score
RandomForestClassifier	**0.92**	**0.91**	**0.91**	**0.91**
ExtraTreesClassifier	0.90	0.89	0.91	0.90
KNeighborsClassifier	0.88	0.87	0.88	0.87
ExtraTreeClassifier	0.87	0.90	0.88	0.89
DecisionTreeClassifier	0.87	0.85	0.83	0.84
SVC	0.85	0.88	0.86	0.87
LogisticRegression	0.80	0.78	0.81	0.80
LinearSVC	0.78	0.75	0.75	0.75
Perceptron	0.68	0.65	0.69	0.67
RNN	0.65	0.64	0.65	0.64

Table 5.3: Results obtained for Lazy Classifier models after running the algorithm on jEdit data using **median value** as a reduction method

Model	Accuracy	Recall	Precision	F1-Score
RandomForestClassifier	**0.90**	**0.91**	**0.91**	**0.91**
ExtraTreesClassifier	0.91	0.90	0.90	0.90
KNeighborsClassifier	0.87	0.86	0.85	0.86
ExtraTreeClassifier	0.87	0.88	0.86	0.87
DecisionTreeClassifier	0.86	0.82	0.80	0.81
SVC	0.85	0.86	0.84	0.85
LogisticRegression	0.79	0.75	0.78	0.77
LinearSVC	0.78	0.74	0.74	0.74
Perceptron	0.67	0.65	0.66	0.66
RNN	0.65	0.64	0.63	0.64

Table 5.4: Results obtained for Lazy Classifier models after running the algorithm on FreeMind data using **average mean** as a reduction method

method, achieving an accuracy of 95% for jEdit, 93% for FreeMind and 94% for TuxGuitar.

Model	Accuracy	Recall	Precision	F1-Score
RandomForestClassifier	**0.93**	**0.92**	**0.91**	**0.92**
ExtraTreesClassifier	0.91	0.90	0.90	0.90
KNeighborsClassifier	0.88	0.87	0.86	0.87
ExtraTreeClassifier	0.88	0.89	0.86	0.87
DecisionTreeClassifier	0.87	0.83	0.81	0.82
SVC	0.86	0.87	0.85	0.86
LogisticRegression	0.80	0.76	0.79	0.78
LinearSVC	0.79	0.75	0.75	0.75
Perceptron	0.68	0.66	0.67	0.67
RNN	0.66	0.64	0.64	0.64

Table 5.5: Results obtained for Lazy Classifier models after running the algorithm on FreeMind data using **sum** as a reduction method

Model	Accuracy	Recall	Precision	F1-Score
RandomForestClassifier	**0.91**	**0.90**	**0.90**	**0.90**
ExtraTreesClassifier	0.88	0.87	0.89	0.88
KNeighborsClassifier	0.86	0.86	0.87	0.86
DecisionTreeClassifier	0.86	0.84	0.82	0.83
ExtraTreeClassifier	0.85	0.88	0.86	0.87
SVC	0.84	0.87	0.85	0.86
LogisticRegression	0.78	0.76	0.79	0.78
LinearSVC	0.77	0.74	0.74	0.74
Perceptron	0.66	0.63	0.67	0.65
RNN	0.63	0.64	0.63	0.64

Table 5.6: Results obtained for Lazy Classifier models after running the algorithm on FreeMind data using **median value** as a reduction method

Model	Accuracy	Recall	Precision	F1-Score
RandomForestClassifier	**0.89**	**0.90**	**0.90**	**0.90**
ExtraTreesClassifier	0.89	0.88	0.88	0.88
KNeighborsClassifier	0.86	0.85	0.84	0.85
ExtraTreeClassifier	0.85	0.86	0.84	0.85
DecisionTreeClassifier	0.85	0.81	0.79	0.80
SVC	0.83	0.84	0.83	0.83
LogisticRegression	0.78	0.74	0.77	0.76
LinearSVC	0.76	0.72	0.72	0.72
Perceptron	0.66	0.64	0.65	0.65
RNN	0.63	0.62	0.63	0.62

Table 5.7: Results obtained for Lazy Classifier models after running the algorithm on TuxGuitar data using **average mean** as a reduction method

Model	Accuracy	Recall	Precision	F1-Score
RandomForestClassifier	**0.94**	**0.91**	**0.91**	**0.91**
ExtraTreesClassifier	0.92	0.90	0.90	0.90
KNeighborsClassifier	0.90	0.86	0.86	0.86
ExtraTreeClassifier	0.89	0.86	0.86	0.86
DecisionTreeClassifier	0.88	0.82	0.81	0.81
SVC	0.85	0.86	0.84	0.85
LogisticRegression	0.80	0.76	0.78	0.77
LinearSVC	0.78	0.74	0.75	0.74
Perceptron	0.67	0.66	0.67	0.67
RNN	0.66	0.65	0.65	0.65

Table 5.8: Results obtained for Lazy Classifier models after running the algorithm on TuxGuitar data using **sum** as a reduction method

Model	Accuracy	Recall	Precision	F1-Score
RandomForestClassifier	**0.90**	**0.89**	**0.90**	**0.89**
ExtraTreesClassifier	0.87	0.86	0.88	0.87
KNeighborsClassifier	0.85	0.85	0.86	0.85
DecisionTreeClassifier	0.85	0.83	0.81	0.82
ExtraTreeClassifier	0.84	0.87	0.85	0.86
SVC	0.83	0.86	0.84	0.85
LogisticRegression	0.77	0.75	0.78	0.77
LinearSVC	0.76	0.73	0.73	0.73
Perceptron	0.65	0.63	0.66	0.64
RNN	0.62	0.62	0.65	0.63

Table 5.9: Results obtained for Lazy Classifier models after running the algorithm on TuxGuitar data using **median value** as a reduction method

5.4 Discussion

A codebase with a high percentage of technical debt issues may be difficult to maintain and may have serious problems with the design, architecture, or implementation of the code. These issues might make it more challenging and time-consuming to add new features or address faults. They can also cause problems of functionality, performance, and reliability of the software system. Hence, the large number of detected issues suggests an increase need of a refactoring in all three studied projects, being in agreement with the refactorings that were performed upon them. The diversity of the issues also is in sync with the large number of refactoring types that were performed, proving that the issues that needed to be solved were diverse, implying different changes and different risks.

The findings of this study can be used in the software development domain for predicting the needed software refactorings by using the proposed web application. However, at least for now, this

app can take its input in two possible ways: it either takes .xlsx files that need to contain a few columns, namely: name of the component, issue's severity, issue's debt and issue's type, or it can be used directly with SonarQube, needing to know the credentials and the name of the project that needs to be analyzed. For the input type being .xlsx files, they can contain other columns, but the existence of the previously mentioned columns is mandatory since the prediction of the number of refactorings and the software maintainability is done based on them. The data on technical debt issues used when building the intelligent model follows the format that the SonarQube [**?**] reports have, so the values of the provided file to the application should also be between the limits proposed by SonarQube. This represents a limitation of the proposed web application since it complicates a little the usage of the application for the situation in which other tools than SonarQube were used for performing the static analysis. If SonarQube was used for static analysis, than this application can connect directly to it and get the needed information for making the requested prediction.

The results of this study are encouraging since the performance of the system is quite high. When treating the problem as a classification problem and computing the metrics specific to this kind of problem, the obtained values for accuracy, recall and F1 Score were high: 0.97 for accuracy and balanced accuracy, 0.96 for recall and F1 Score.

Chapter 6

SWOT Analysis

There are many aspects that need to be analyzed when building a web application that detects the types software refactoring processes needed by each software component, this prediction being made based on technical debt issues data. Because this is a topic coming from an area which receives more and more interest, it is important to cover as many factors that influence the behavior of the application as possible. Firstly, the classification model needs to be efficient enough. Secondly, the integration of the classification model is very important, because the entire flow of the final app depends on that.

6.1 Strengths

When talking about the strengths of the application, those can be divided into two categories: capabilities of the classification algorithm, and capabilities of the final application (web application in which the classification model was integrated).

When referring to the first category, the high accuracy of the final model, 95%, represents such a strength, showing that the classification model correctly classifies almost all types of considered software refactoring opportunities given to it. Another strength is represented by the fact that the prediction is done based on technical debt issues data, there being many studies showing the strong relationship between refactoring opportunities and technical debt issues. One important aspect that needs to be mentioned here is that the prediction is done based on all technical debt issues (code smells, bugs and vulnerabilities) and not only code smells, as this brings a larger spectrum of information. Regarding the final application, it is important to mention that the application is a Flask web application, which can be easily used by either uploading a report file from the static analysis or directly connecting to the SonarQube for obtaining the needed data. Also, the application has a straight-forward flow, and is easy-to-use.

6.2 Weaknesses

The main weakness is represented by the fact that currently, the classification model classifies only 28 refactoring opportunities, 27 refactoring types and the scenario in which no refactoring is needed at that moment, while there are far more than that.

Another limitation is represented by the time the application needs to make the prediction, time that could be improved in the future. Besides those, the performance obtained in the second approach, the one using the RNN model, having more than one entru per each software component is not satisfactory. This is the reason for which this model was not chosen to be integrated into the final application, even if the data that was feed to it is the most comprehensive since there is no risk that important data might be lost during the reduction process.

In addition to the limitations described above, the fact that the user needs to connect explicitly to the app online might represent a minus, since it implies that the user enters the browser to connect to the application, and cannot do that directly from their IDE or from SonarQube.

6.3 Opportunities

With respect to the detection of refactoring opportunities, this process can be improved from multiple directions. The first one is represented by enriching the dataset, adding multiple refactoring types and more data about the currently supported refactoring types, in order to balance the dataset. Another aspect is represented by increasing the performance of the RNN approach, as this approach considers multiple aspects from the given data.

Also, a future direction that would prove itself useful for the users would be to integrate this app as a SonarQube plugin. This way, the detection of software refactorings based on technical debt issues data could be easily accessed from every static analysis done in SonarQube.

6.4 Threats

The principal threat is represented by the possibility of incorrectly detecting a software refactoring opportunity. Since the accuracy is not 100%, this possibility is valid. This would have a negative impact as if an engineer performs a software refactoring that is not needed, instead of increasing the quality of the code, it might over-complicate it resulting in a lower maintainability hence quality. Another threat might be represented by the small numbers of refactoring types considered in this application. The app might suggest a certain refactoring, which is considered suited only because a

better match was not studied yet. Also, the model was trained only on Java code, and that might represent a threat since its generalization might be questioned.

Another threat might be represented by the security of the application. The current version of the application does not posses a strict security policy. For the moment, the final application is a simple, regular Flask web application which has a .sav model integrated, used for detecting software refactoring opportunities from technical debt issues reports, and it does not have any addition data protection.

Chapter 7

Conclusion and future work

The evolution of software systems is a phenomenon that has become a constant in the past decades. In order to detect methods of improving the maintainability of software systems, the analysis of software source code has received more and more attention lately, using the growth of the artificial intelligent segment to better analyze specific problems. Many studies have shown that there is a strong connection between software maintainability, technical debt issues and code refactorings.

In this paper, different approaches on detecting software refactoring opportunities based on technical debt issues were studied, in order to find a high performing system which is also suitable for web integration.

In order to accurately predict the software maintainability and the number of needed code refactorings, a comprehensive dataset was needed. In this study, three open-source Java projects were studied, namely jEdit, FreeMind and TuxGuitar. For obtaining the technical debt issues' data, static analysis was performed on the following versions of the projects: jEdit 5.5, FreeMind 1.0.1 and TuxGuitar 1.5.2. For obtaining the refactoring data, the RefactoringMiner tool was used to compare the versions 5.5 and 5.6 of jEdit, 1.0.1 and 1.1.0 of FreeMind and 1.5.2 and 1.5.3 of the TuxGuitar project. The technical debt data from [7] was combined with the data obtained after using the RefactoringMiner tool, and was further processed. Two different approaches have been considered. The first approach compressed the technical debt issues to only one per each software component, and added as another feature the number of technical debt issues detected for each software component. After reducing the technical debt issues to only one per software component, data augmentation was performed in order to balance the dataset. This approach used as an intelligent algorithm a few classical machine learning algorithms, among which the Decision Trees algorithms, Random Forest algorithms, Support Vector Machine and others. The best results, for both the classification and the regression task, were obtained by the ExtraTrees algorithm. This obtained an accuracy of 0.95 and a F1-Score of 0.93. The second

approach didn't modify the technical debt issues associated to each software components, meaning that the same component could have had more than one entry in the dataset. For this approach, a RNN was used as the intelligent algorithm, but the model didn't perform as well as the ExtraTrees algorithm from the first approach, obtaining only an accuracy of 0.66. The best performing model was saved as a .sav model, and was further used in a Flask web application that allows the user to predict the refactoring opportunities of each software component, either by directly uploading the results of the static analysis, or by linking the web application to SonarQube.

The proposed methodology utilizes technical debt data to detect the software refactoring opportunities across three open-source Java projects. It is important to acknowledge a potential threat to the validity of the obtained outcomes, namely that the utilized projects were written in Java. Consequently, the results and the model's performance may not be as reliable when applied to projects developed in other programming languages. Furthermore, it is worth exploring certain aspects that were not taken into account within the current approach, such as specific details related to technical debt issues (e.g., message content) or refactorings (e.g., description and number of refactorings). Incorporating the message content of the issues and the type or description of refactorings into the prediction model may improve accuracy and provide more insightful information to the end user. Additionally, the current implementation solely relies on technical debt data, and a prospective future enhancement to the study would involve considering additional software metrics. Such metrics offer valuable insights into the code's state and their inclusion could enhance the predictive model. However, those metrics would be useful only in the first approach, as they are common per each software component, and introducing them in a model that has several entries per component might complicate the learning process leading to a low performance. Another direction worth exploring refers to refining the performance of the second proposed approach, as it does not condense technical debt data to a single entry per software component. Likewise, in the first approach, the reduction of technical debt data to a single entry per component is achieved by calculating either the average mean, the sum or the median of all values. However, it is crucial to note that there might be more optimal reduction methods. This presents an additional area warranting further investigation.

The obtained results represent a step forward into realizing a high performing detection of software refactoring opportunities, which can also be integrated into a web application, in order to be easily accessible. This might help the software community to better assess software maintainability, reducing the resources needed for the maintenance phase.

Bibliography

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

[2] Abdulrahman Ahmed Bobakr Baqais and Mohammad Alshayeb. Automatic software refactoring: a systematic literature review. *Software Quality Journal*, 28(2):459â502, June 2020.

[3] Neil A. Ernst and David A. Eichmann. The future of software maintenance. *IEEE Software*, 16(1):44–50, 1999.

[4] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, USA, 1999.

[5] Peter Hegedus, Istvan Kadar, Rudolf Ferenc, and Tibor Gyimothy. Empirical evaluation of software maintainability based on a manually validated refactoring dataset. *Information and Software Technology*, 95:313–327, 2018.

[6] IEEE. Ieee standard glossary of software engineering terminology, 1990. IEEE Std 610.12-1990.

[7] Arthur-Jozsef Molnar. Collection of technical debt issues in freemind, jedit and tuxguitar open source software, 2020.

[8] Nikolaos Tsantalis, Ameya Ketkar, and Danny Dig. Refactoringminer 2.0. *IEEE Transactions on Software Engineering*, 48(3):930–950, 2022.

[9] Liming Zhao and J Hayes. Predicting classes in need of refactoring: an application of static metrics. In *Proceedings of the 2nd International PROMISE Workshop, Philadelphia, Pennsylvania USA*, 2006.