Winter 3-17-2023

Predicting code refactoring via analyzing the history of quality metrics and code anti-patterns

Sarah Alanqari
DePaul University, salangar@depaul.edu

PREDICTING CODE REFACTORING VIA ANALYZING THE HISTORY OF QUALITY METRICS AND CODE ANTI-PATTERNS

BY

SARAH MOHAMMAD F ALANQARI

A THESIS SUBMITTED TO THE SCHOOL OF COMPUTING, COLLEGE OF COMPUTING AND DIGITAL MEDIA OF DEPAUL UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE IN SOFTWARE ENGINEERING

DEPAUL UNIVERSITY

CHICAGO, ILLINOIS

2023

DePaul University
College of Computing and Digital Media

MS Thesis Verification

This thesis has been read and approved by the thesis committee below according to the requirements of the School of Computing graduate program and DePaul University.

Name: Sarah Mohammad Fahd Alanqari

Title of dissertation: Predicting Code Refactoring via Analyzing the History of Quality Metrics and Code Anti-Patterns

Date of Dissertation Defense: March 17, 2023

Wael Kessentini

Advisor*

Vahid Alizadeh

1st Reader

Zhen Huang

2nd Reader

** A copy of this form has been signed, but may only be viewed after submission and approval of FERPA request letter.*

Dedication

I dedicate this to God, who has helped me on my life's journey. To my parents, my brother, and my friends, for all their endless support, and encouragement.

Acknowledgments

I want to take a moment to express my gratitude to everyone who has supported me During my thesis.

First and foremost, I want to express my sincere gratitude to the Almighty God for giving me the ability to accomplish my master's thesis and degree.

To my adviser, Dr. Wael Kessentini, I would like to convey my deepest gratitude for his help, ongoing advice, and tremendous support for doing this research.

In addition to my adviser, I want to thank Dr. Thiago Ferreira for his insightful feedback and suggestions for my thesis.

I also want to express my gratitude to the Department of Software Engineering teachers for exposing me to various software engineering disciplines over the course of my Master's degree.

For financial support, I'd like to thank the Saudi Arabian Cultural Mission (SACM) for helping me financially and allowing me to study for a master's degree.

Last but not least, I must thank my parents for spiritual support and my brother who devoted time to staying and accompanying me in the US to complete my master's study.

Abstract

Code refactoring is the process of improving the internal structure of existing code without altering its functionality. Refactoring can help to reduce technical debt, enhance the quality of the code and make the code easy to evolve. However, the manual identification of the proper code refactoring operations to apply can be time-consuming and not scalable.

In this thesis, we propose an approach based on data mining and machine learning techniques to analyze historical data and predict refactoring operations that may occur in a future release of a project. The approach uses a combination of techniques to identify patterns in the data and make predictions about which refactoring operations should be applied. In this study, we validated the proposed machine learning based approaches with 13 open-source projects with different releases. We identified the refactoring operations and code smells and extracted the quality metrics for each project release. We used the collected data (e.g. quality metrics and code smells) to predict refactoring operations, and we reported the prediction results based on cross-validation procedures.

The proposed research contributes to the field of software quality by providing an efficient and effective approach to refactor the code. The findings of this research will also help developers by suggesting appropriate refactoring operations based on the history of the evolution of software projects. This will ultimately result in improved software quality, reduced technical debt, and enhanced software performance.

Keywords: Software quality, Software maintenance, Refactoring prediction, Software Evolution

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Software systems inevitably require maintenance and modification in response to market changes and customer needs, such as bug fixes, modified requirements, new features, etc. The success of a software system depends on its ability to retain a high-quality design in the face of continuous change [1] Developers use different techniques, such as refactoring, to prepare software systems for further modifications.

Refactoring is the process of improving the code's structure without changing its external behavior by conducting some modifications [2]. Developers need the refactoring process, especially if there is a significant expansion and modifications in the code or when the project turns from a small or individual project to a large or group project. To guide their refactoring efforts, developers can identify areas of the codebase that require improvement using quality metrics or by detecting code smells in the source code. Quality metrics are measurements used to evaluate the quality of software systems [3]. The developers can use these metrics to assess various aspects of the software, such as its performance, reliability, maintainability, usability, etc. Quality metrics are essential for evaluating the effectiveness and impact of refactoring on software systems [4]. On the other hand, code smell is a sign of potential design issues in the codebase, indicating that part of the code may be difficult to understand, maintain, or modify in the future. Code smells are warning signs that part of the code may require refactoring.

However, finding the appropriate refactoring operations to apply to a code can be challenging because it requires extensive knowledge and experience. The same piece of code can be subjected to various refactoring operations, but picking the best ones requires knowledge of the codebase and the environment in which it runs.

Machine learning and data mining techniques have recently gained popularity in software engineering research to support decision-making processes [5]. The purpose of this thesis is to investigate the use of machine learning algorithms in predicting appropriate refactoring operations. The research will involve:

- Analyzing existing codebases and identifying common patterns and relationships within the code.
- Preprocessing the data to create features like code smells and quality metrics that can be used to train machine learning algorithms.
- Evaluating several machine learning algorithms, and the most suitable algorithm will be selected based on its accuracy and performance.

The proposed research will contribute to the field of software engineering by providing a more efficient and effective approach to refactor the code. The findings of this research will also help developers by suggesting appropriate refactoring operations. This will ultimately result in improved software quality, reduced technical debt, and enhanced software performance.

The remainder of this thesis is organized as follows. Chapter 2 states some related works, including code smells, quality metrics, and refactoring. Chapter 3 describes our research methodology to predict the refactoring operations. Chapter 4 discusses the results of our thesis. 5 clarifies the conclusion and plans for future work.

Chapter 2

Related Work

In this chapter, we first provide the necessary background and discuss the correlation between code smells, quality metrics and refactoring. Also, we give an overview of the state of the art related to refactoring prediction.

2.1 Code Smells

Code smells, also called design anomalies or design defects, refer to design situations that adversely affect software maintenance [3]. In general, they make a system difficult to change, which may introduce bugs. Different types of code-smells, presenting various symptoms, have been studied to facilitate their detection and suggest improvement solutions [8]. Sjoberg et al. [13] discussed the impact of code smells on software maintenance efforts in an industrial setting. Deligiannis et al. [14] conducted a study to understand how God Classes affected the software's maintainability. Their methodology showed that maintainability and understandability increase with design quality. They also concluded that God Classes were detrimental to the code's quality. In addition, Olbrich et al. [15] investigated how two types of smells, called God Class and Brain Class, affect the quality of software systems. Their research then suggests further investigation to ensure the results can be generalized and looks at the correlation between the size of the system and the number of classes. In addition, they explained how to avoid these types of smells. Moha et al. [10] have proposed a taxonomy of design smells that describes the structural relationships between the codebase and code smells and their measurable, structural, and lexical properties. It also describes the structural relationships between design and code smells. Zhan et al. [16] conducted a systematic literature review where the results of 39 studies on code smells were synthesized. The review also discussed the related problems of identifying code smells and detecting refactoring opportunities. Another systematic literature review was conducted by Al Dallal [18], which focused on providing an overview of code smell identification approaches. The review analyzed 47 studies and concluded that while there has been a significant increase in identifying refactoring opportunities, the focus was on identifying code smells. In their analysis, Santos et al. [21] examined 64 studies published from 2000 to 2017. One of their significant findings was the lack of reliability in human-based evaluation of code smells. They observed a trend in the selected studies that showed a low level of consensus among developers in detecting smells. Additionally, on their analysis of the selected studies, the authors noted that demographic data, such as a developer's experience, could significantly influence code smell evaluation based. Fernandes et al. [30] conducted a systematic literature review focused on the tools available for detecting code smells. Their study aimed to identify and analyze the main features of code smells detection tools, as well as the types of code smells they can detect.

Through these studies, it is clear that the code smells negatively affect the quality of the source code, as their presence may lead to the need to maintain the code with more effort and time.

2.2 Quality Metrics

Software quality metrics are measurements used to evaluate software systems' quality and assess various aspects of the software such as security, maintainability, reliability, etc.

Many researchers have proposed different metrics for assessing software quality, such as measures of code complexity, maintainability, reliability, performance, security, and other aspects of software quality. For instance, Chidamber and Kemerer's CK metrics suite [31] is a well-known set of measures for evaluating object-oriented software. Similarly, Halstead's metrics [23] and McCabe's cyclomatic complexity [26] are popular metrics for assessing code complexity. Moreover, Ganjare et al. [24] stated that software developers could benefit from an early estimation of their product's quality. This is because field quality information is often only available too late in the software lifecycle. It is important to identify early indicators of a product's quality. This can help reduce costs associated with the software and the amount of time spent on maintenance and upkeep. To measure code quality, extra information is taken from various software engineering processes, and quality metrics are used. Poor quality software will often result in low accuracy results after testing. Software readability, reliability, maintainability, and reusability can all be calculated based on the values of these metrics. Using this information, the quality of the code can be improved, and the costs associated with maintenance can be reduced.

These studies show that quality metrics are important in the software maintenance process as they help developers discover and fix problems during maintenance and reduce associated costs. Also from these studies, we learn about the existing software metrics extraction tools.

2.3 Refactoring

Refactoring is the process of improving the design and structure of existing code without changing its behavior or functionality. It involves making small, incremental changes to the codebase to improve its readability, maintainability, and extensibility, and to reduce technical debt. [3]. Refactoring can be done using refactoring operations. Refactoring operations are actions used to modify existing code in a software to improve its overall quality. Refactoring operations typically do not change the functionality of the code but rather focus on improving its internal structure [3]. Although refactoring may help improve the code, it may cause a negative effect, for example, it may increase the code smells on the code.

Cedrim et al. [17] presented the findings of a study that aimed to explore the relationship between code refactorings and code smells. Their study states that refactorings do not always have the desired effect and can sometimes introduce new code smells. Their study also explains that by looking at recurring refactoring-smell pairs, the study was able to identify non-removal, removal, and creational patterns and found that extract method refactorings were frequently involved. Finally, their study suggests that this information can help practitioners and tool engineers because they can better inform developers when refactorings may produce stinky or neutral effects and a refactoring assistant tool can help developers by informing them when refactoring may be introducing a code smell and suggesting how to fix it. This study helped us to understand the correlation between refactoring and code smells.

Also, refactoring may affect the quality of the program. For instance, Hamdi et al. [22] aimed to study the effects of refactoring on the quality metrics of Android apps. They chose to look at 300 open-source apps containing over 42,000 refactoring operations. The authors wanted to see how these refactoring operations impacted the ten selected software quality metrics. To measure this change, they used the difference-in-differences (DiD) model. The results of the study showed that refactoring did have an impact on the metrics, but it was not as clear-cut as they had initially expected. Some types of refactoring improved the metrics, while others worsened them. Alomar et al. [34] empirically analyzed the impact of refactoring operations on a set of common state-of-the-art design quality metrics. The study involves extracting a large corpus of design-related refactorings applied in 3,795 curated open-source Java projects. The researchers identified 1,245 quality improvement commits and analyzed the impact of the corresponding refactoring operations on a set of state-of-the-art design quality metrics. The statistical analysis revealed that some metrics are more popular and emphasized than others. The study confirms that various structural metrics can represent internal quality attributes with varying degrees of improvement and degradation of software quality. Additionally, while most metrics mapped to quality attributes capture developer intentions of quality improvement, some do not.

The studies have contributed to our understanding of the impact of refactoring operations on quality metrics, as they have shown that these measures can be positively or negatively affected. Based on these findings, there has been interest in using these metrics to predict refactoring operations.

2.4 Refactoring Prediction

Several studies have shown that refactoring operations can be predicted. Ratzinger et al. [11] tried to predict locations of future refactoring based on the development history. The study aimed to investigate whether attributes of software evolution data could be used to predict the need for refactoring in the next two months of development. The researchers utilized information systems in software projects to extract data mining features such as growth measures, relationships between classes, and the number of authors working on a particular piece of code. They used this information as input into classification algorithms to create prediction models for future refactoring activities. Sagar et al. [12] conducted a study to determine if machine learning and neural network models can accurately predict the refactoring class of any project. The first experiment used only the commit messages as input, but the results were not good enough. The second experiment attempted to combine the commit messages with code metrics, and the accuracy increased to 54.3%. A third experiment attempted to use 64 different code metrics related to cohesion and coupling and this improved the accuracy to 75%. The study results suggest that code metrics are better for predicting class refactoring than commit messages alone, as the latter have a limited vocabulary and are not enough for training ML models.

In addition, Kumar et al. [35] The study aims to develop a recommendation system for identifying code segments that require refactoring. The approach uses 25 source code metrics at the method level as features in a machine learning framework to predict the need for refactoring.

Similarly, Aniche et al. [36] investigated the effectiveness of machine learning algorithms in predicting software refactoring using quality metrics as a feature.

Furthermore, Akour et al. [37] developed a tool that uses a class-based approach to predict which classes require refactoring based on a set of static metrics and a weighted ranking method. They conducted a study comparing the tool's performance to human reviewers and found that it effectively identified design problems that reduce class maintainability. The study also provided insights on how to enhance the functionalities of the tool.

Through these studies, we have a background on the methods used by researchers to predict refactoring operations.

In conclusion, the studies discussed in this chapter provided valuable insights on detecting refactoring needs, highlighting the significance of refactoring for enhancing software quality and demonstrating its effects on quality metrics and code smells. Additionally, we presented different approaches recently focusing on predicting refactoring.

Chapter 3

Methodology

In this chapter, we describe the methodology used in this research. The research involves the following steps described in Figure 1.

In the first phase, we collected the data for analysis from open-source software repositories. In the second phase, the data is processed to create features that can be used to train machine learning algorithms. The preprocessing step involves feature extraction, using different tools to extract refactoring operations, code smells and quality metrics, and feature selection to select the most suitable set. The last phase involves the machine learning algorithm selection, model training and evaluation, and result analysis.

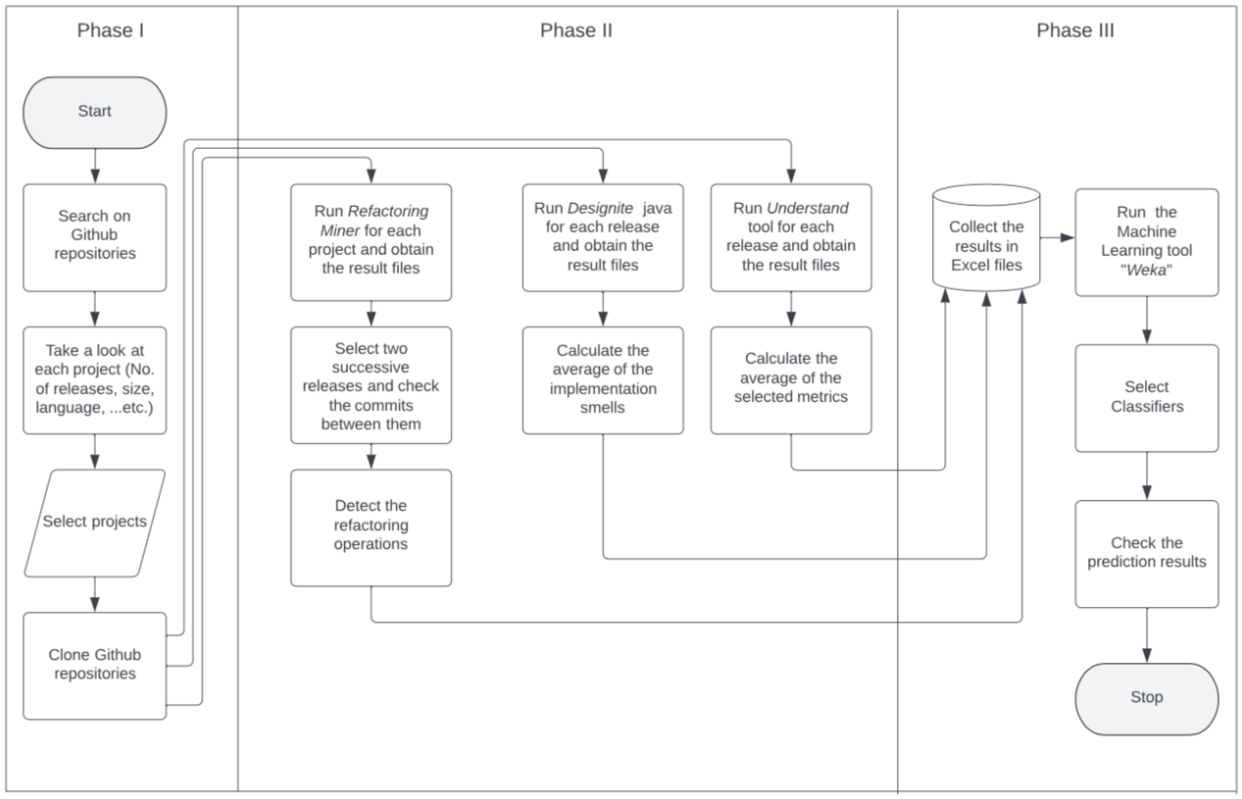

Phase I
Phase II
Phase III
Start
Search on Github repositories
Take a look at each project (No. of releases, size, language, ...etc.)
Select projects
Clone Github repositories
Run Refactoring Miner for each project and obtain the result files
Select two successive releases and check the commits between them
Detect the refactoring operations
Run Designite java for each release and obtain the result files
Calculate the average of the implementation smells
Run Understand tool for each release and obtain the result files
Calculate the average of the selected metrics
Collect the results in Excel files
Run the Machine Learning tool "Weka"
Select Classifiers
Check the prediction results
Stop

Figure 1: Approach Overview.

3.1 Data Collection

In this section, we describe the phase of data collection (Phase 1 in Figure 1). It consists of selecting the projects to be studied and the criteria for selecting them such as the size of the source code, its version history, etc. For mining of software projects, we used Github repositories which provide details about the project sizes, commits, etc. We looked through 9,208 random open-source Java projects from a list of GitHub repositories. We only searched for Java projects because it is the only language supported by some of the tools that we will use. We manually checked the first 100 generated pages throughout the resulting search pages. While scanning the projects, we eliminated those with no releases. As a result, we ended up with ten projects. The projects were chosen by ranking them according to the size of the project, the number of versions, number of commits, and the number of commits. To ensure the diversity and quality of our data, we added three other projects used in a study [7] that tried to find the relationship between refactoring operations and code smells. Each project has at least 6 versions. Table 1 shows details of collected projects.

3.2 Feature Extraction and Selection

The next step is to extract relevant features from the collected data. This involves identifying refactoring operations and extracting quality metrics, code smells features that can be used as input for the machine learning algorithms.

3.2.1 Refactoring Detection

To identify the whole refactoring history of each project, we used a tool called *Refactoring Miner*, developed by Tsantalis et al. [6]. *Refactoring Miner* is made to examine code commits in software repositories to find any refactoring that has been used. The tool works by analyzing the history of a software project and identifying changes that correspond to refactoring. First, it identified all the changes made to the source code between two consecutive versions of the software repository, such as commits or revisions. Then, it analyzes the changes to detect whether they represent a refactoring. This analysis involves comparing the code before and after the modifications and searching for specific patterns that indicate a refactoring. Once a potential refactoring is detected, *Refactoring Miner* applies a set of rules and heuristics to validate whether the change corresponds to a refactoring. If the change meets the criteria for refactoring, the tool records it. We decided to use this tool because it has shown promising results in detecting refactorings compared to the state-of-the-art tools [38].

We run the tool for each project of our collected dataset to obtain all the refactorings that happened in the entire project. We focused solely on releases containing refactorings to prepare the training dataset. To ensure this, we manually examined the results of every two consecutive releases and retained only those that included refactorings (Phase 2 in Figure 1).

We selected 20 refactoring operations to use in our experiments. Some of these refactoring operations are among the most popular refactoring types according to a prior study [39] [40]. The list of the selected refactoring operations is described in Table 2.

3.2.2 Code Smells and Quality Metrics Detection

To extract code smells from each project release, we used a tool called *Designite* developed by Sharma et al. [19]. This tool takes the source code of a project and produces a dataset of code smells at various levels, including implementation, design, and architecture. *Designite* is compatible with both C# and Java, and since our chosen projects were developed in Java, we opted for this tool. Additionally, *Designite* proved to be an efficient and straightforward means of generating datasets of code smells.

Table1: Projects Overview.

Projects	No. of lines	No. of commits	No. of refactorings	Versions
datacleaner/DataCleaner	231857	5693	39598	5.2.2, 5.3.0, 5.4.0, 5.4.1, 5.4.2, 5.5.0, 5.6.0
kinhong/OpenLabeler	218806	53	2535	v1.0.0, v1.0.1, v1.1.0, v1.1.1, v1.1.2, v1.2.0, v1.2.3, v1.2.4
aws/aws-toolkit-eclipse	149928	151	4242	2.7.1, 2.8.0, 2.8.1, 2.8.2, 2.9, 2.9.1, v201706150006
TheCoder4eu/BootsFaces-OSP	146241	1866	3731	v0.5.0, v0.6.0, v0.6.5, v0.6.6 , v0.7.0 , v0.8.0
bioinformatics-ua/dicoogle	108277	1030	2157	2.0.0, 2.1.0, 2.2.0, 2.3.0, 2.3.1, 2.4.0, 2.5.0
QuantumBadger/RedReader	99946	2076	7119	v1.9.6, v1.9.6, v1.9.7, v1.9.8, v1.9.8.1, v1.9.8.2
Etar-Group/Etar-Calendar	79475	3100	3126	v1.0.1, v1.0.8, v1.0.9, v1.0.10, v1.0.11, v1.0.12, v1.0.13
ThirtyDegreesRay/OpenHub	54625	495	1188	v1.0.0, v1.0.1, v1.0.2, v1.1.0, v1.1.2, v1.1.3, v1.2.0, v1.2.1
graphhopper/jsprit	54327	2066	8273	1.7-RC, v1.7, v1.7.1, v1.7.2, v1.7.3, v1.8
oliexdev/openScale	43.679	1850	2772	v1.0, v1.1, v1.2, v1.3, v1.3.1, v1.4, v1.4.1
facebook/fresco	1207424	3781	9531	v0.1.0, v0.2.0, v0.3.0, v0.4.0, v0.5.0, v0.5.1
Netflix/Hystrix	78964	1543	5658	1.3.0.RC1, 1.3.0.RC2, 1.3.0, 1.3.1, 1.3.2, 1.3.4, 1.3.5, 1.3.6
PhilJay/MPAndroidChart	44550	1740	5769	v1.0.1, v1.1.1, v1.2.1, v1.3.1, v1.3.2, v1.4.0

In addition, we gathered data on code smells for each individual release using *Designite* (Phase 2 in Figure 1). Running the tool on the selected release generated a database of code smells that were detected within implementation files. Subsequently, we computed the average of occurrences for each code smell in every release. Out of all the code smells identified, nine were found to be present in nearly all of the releases for the selected projects. Therefore, we opted to focus on these nine code smells for further analysis.

The list of the selected code smells detected by *Designite* java tool is described in Table 3.

Table 2: Refactoring Types and Their Descriptions [3].

Refactoring Type	**Description**
Inline Method	Put a method's body into its callers' bodies once it is equally as clear as its name, then remove the method.
Move Class	Move all the features of a class that isn't doing very much into another class and delete it.
Rename Package	If the package's name does not make clear what it is for, change it.
Rename Method	If the method's name does not make clear what it is for, change it.
Move Method	A method uses or will use more features of a different class than the class it is defined on. Move it there.
Pull Up Method	You have methods on subclasses that produce identical results. Transform them into the superclass.
Pull Up Attribute	If the same attribute is shared by two subclasses, transform the attribute into the superclass.
Extract Superclass	There are two classes that have similar features. Make a superclass, then add the shared features to it.
Push Down Method	Only some subclasses can benefit from the behavior of the superclass. Transfer it to those classes.
Push Down Attribute	A field is only used by a specific subclass. To those subclasses, move the field.
Add Parameter	A method requires its caller to provide extra details. For an object that can pass these details, add a parameter.
Add Attribute Modifier	An attribute needs to add a modifier to adjust the corresponding attribute.
Add Method Annotation	An annotation to a method offers details that can be used during runtime or compilation to carry out additional processing for this method.
Add Class Annotation	An annotation to a class offers details that can be used during runtime or compilation to carry out additional processing for this class.
Inline Variable	If a variable is only given the outcome of a simple expression, the references to the variable should be replaced with the expression itself.
Change Parameter Type	The type of a parameter does not match its returned type. Change the type of the parameter.
Extract Variable	If an expression is complex, make it more understandable, by creating a local variable and replacing all occurrences of this expression.
Rename Variable	Change the name of the variable if its name does not reveal its purpose.
Change Attribute Type	The type of an attribute does not match its returned type. Change the type of the attribute.
Change Variable Type	The type of a variable does not match its returned type. Change the type of the variable.

Table 3: Code Smells and Their Descriptions.

Code Smells	**Description**
Complex Conditional	Large condition blocks (if-then/else or switch) that are hard to follow
Complex Method	Methods that are hard to understand
Empty catch clause	The catch block has nothing to do
Long Identifier	Identifiers that have many characters
Long Method	Methods that have many tasks to do
Long Parameter List	Methods that have more than three/four parameters
Long Statement	Statements contain too many lines of code
Magic Number	The direct usage of numbers
Missing default	Expressions with several conditions misses a default case

During this phase, we also detected and analyzed the quality metrics (Phase 2 in Figure 1). To achieve this, we used a tool called *Understand* [20] which enables developers to gain a deeper understanding of their source code and document it effectively. We decided to use this tool because it is compatible with the Java language and it offers clear export of quality metrics. We gathered data on the quality metrics for each release individually.

The tool *Understand* can export a lot of quality metrics, but we decided to select only the Chidamber & Kemerer metrics (CK metrics), because they have a strong correlation with software defects [31] [32]. The exported files from the *Understand* tool provide metrics for different entities such as classes, packages, methods, and files. To ensure the accuracy of our results, we only considered the lines that contained values for each metric and discarded the lines that did not have any value. We then calculated the average for each Quality Metric of interest. The chosen quality metrics are listed and described in Table 4.

Table 4: Quality Metrics and Their Descriptions.

Quality Metrics	**Description**
SumCyclomatic	It describes how many linearly independent routes there are through the source code of a program. It is known as Weighted Methods per Class.
MaxInheritanceTree	It measures the greatest distance possible inside a class hierarchy between a node and the root node. It is known as Depth of Inheritance Tree.
CountClassDerived	It determines the number of direct subclasses that are under a class in the hierarchy of classes. It is known as Number of Children.
CountDeclMethodAll	It is the overall number of methods that might be run as a result of a message that an object of a class receives. It is known as Response For a Class.
CountClassCoupled	It is a total number of classes connected to a specific class. It is known as Coupling Between Objects.
PercentLackOfCohesion	It is a measurement of how many method pairs in a class are unconnected, signifying separate, unrelated sections. It is known as Lack of Cohesion of Methods.

3.3 Data Processing and Model Training

The next step in our methodology is to process the collected data, select the appropriate machine learning algorithms to predict the need for refactoring operations based on the selected features, and train the selected machine learning algorithm on the prepared dataset using cross-validation.

3.3.1 Data Processing

The training set is an important component of the machine learning process because it is used to adjust the model's parameters so that it can learn to recognize patterns in the data and make accurate predictions.

After collecting the data of the refactoring operations, code smells, and quality metrics, we transformed the data into a format that can be used in the training. Then, we entered this training set which is an excel file containing the code smells data (or the quality metrics data) for all the projects and releases, one refactoring operation with a value (“yes” or “no”) indicating whether the next release would undergo the refactoring operation.

3.3.2 Choosing the Classifier

The model selection phase aims to choose an algorithm that can accurately capture the patterns and relationships in the data.

Choosing the right classifier for optimal classification was not an easy task. Different algorithms have different strengths and weaknesses. The choice of the algorithm depends on the nature of the data and the problem to be solved. In our study, we used a tool called *Weka* (Waikato Environment for Knowledge Analysis) [25].

WEKA is a popular open-source software for data mining and machine learning tasks. The tool offers various machine learning algorithms and techniques, including decision trees, random forests, neural networks, etc. Additionally, it provides a user-friendly graphical interface that facilitates usage. Since *Weka* offers numerous classifiers, we conducted a search to identify the specific classifiers that would be suitable for our data.Ex In several studies [27] [28] [29], the distinct working mechanisms of various classifiers have been discussed. Each classifier uses a unique approach to distinguish between data. In this thesis, we focused on three classifiers, described below:

- **Random Forest**

  Random forest is an algorithm used for classification as well as for regression. During the training phase, several decision trees are constructed. It then combines the predictions of each decision tree to determine a more accurate overall prediction. It is one of the most accurate and efficient learning algorithms, being able to handle large datasets with many inputs without needing to delete any of them. Furthermore, it is also able to provide estimates of which variables are the most important in the classification. This is done by using a technique known as bagging, which creates random subsets of data and then creates decision trees from these subsets. Finally, the result of the random forest algorithm is determined by looking at the results of each decision tree and taking the prediction that appears the most.

- **Naive Bayes**

The Naive Bayes algorithm is a type of classifier that is used to classify data based on computing and consolidating the frequencies of certain values in a data set. It assumes that all of the attributes in a data set are independent of one another, given the value of the class variable.

- **J48**

J48 is an algorithm designed to help stratify data in order to categorize it. This is done by creating a binary tree that takes each input to determine which category it belongs to. J48 is especially useful because it is able to handle missing values, pruning of decision trees, and continuous values.

3.3.3 Collecting the Results

After entering *Weka* inputs, which are the training sets, for each of the refactoring operations, we applied the chosen classifiers to them one after the other. These inputs are different from each other based on the number of releases and the value of (y/n) of the refactoring operation in the next release.

After applying each classifier to the training sets for each of the refactoring operations, we collected several evaluation measures, including Precision, Recall, and F-measure. These evaluation measures are defined based on the number of true positives (TP), false positives (FP), true negatives (TN), and false negatives (FN) in each classification. TP represents the number of correct predictions made by the model. FP occur when an instance is incorrectly classified as positive when it should not, i.e., the number of instances that were predicted as positive but were actually negative. TN represents the number of correctly predicted negative instances, i.e., the number of instances that were predicted as negative) and were actually negative. FN occurs when an instance is incorrectly classified as negative when it should have been positive, i.e., the number of instances that were predicted as negative but were actually positive. The definitions of the evaluation measures as follow [33]:

- **Precision**

  Precision means the percentage of the number of true positives (TP) divided by the total number of positive results (TP and FP), where 1.0 is the best precision while 0.0 is the worst.

$$\text{Precision} = \frac{\text{TP}}{TP + FP} * 100$$

- **Recall**

  Recall means the percentage of the number of true positives divided by the number of all instances that should have been identified as positive.

$$\text{Recall} = \frac{\text{TP}}{TP + FN} * 100$$

- **F-Measure**

  It is a measurement of a test's accuracy where it is calculated from the test's recall and precision.

$$\text{F} - \text{Measure} = \frac{2 * \text{TP}}{2 * TP + FP + FN}$$

With the different dataset, features and for each refactoring operation, all Precision, Recall, and F-measure scores were collected one by one in an Excel file. Then, these results were considered and analyzed and discussed in the next chapter.

The arrangement of sections in this chapter follows the methodology we adopted in our thesis. Initially, we selected projects and conducted an analysis to identify the presence of refactoring operations, code smells, and quality metrics for each release. Subsequently, we processed this data by creating inputs for *Weka*, selecting classifiers, and obtaining results. In the next chapter we will present and analyze the results to determine the optimal feature and machine learning algorithm for predicting refactoring operations.

Chapter 4
Validation

This chapter is devoted to assessing our approach's performance for refactoring prediction. To achieve this task, we conducted a set of experiments based on different projects with different releases extracted from software repositories. In this chapter, we start by presenting our research questions, and then we analyze and discuss the obtained results.

4.1 Research Questions

The study was conducted to quantitatively assess the completeness and correctness of our refactoring prediction approach. More specifically, we aimed to answer the following research questions (*RQ*):

- ***RQ1:* What impact of the dataset size have on the prediction of refactoring?**

  To answer *RQ1*, we generated different ML models based on a dataset containing projects with different releases. Per each execution, we vary the number of releases in the database. Starting with a dataset with 13 projects and one release per project, until 5 releases per project. The goal is to check the impact of the data size on the accuracy of the prediction and to check if it is possible to predict refactorings with a short project's history (in terms of project releases).

- ***RQ2:* To what extent can the proposed approach predict refactoring operations?**

  To answer *RQ2,* we used the collected data as explained in the previous chapter and generated the results using a machine learning tool. In practice, some prediction algorithms perform better than others, depending on the task. In this RQ, we explore how accurate different ML algorithms are in predicting refactoring operations.

Furthermore, we used different features to train the ML models. In this *RQ* we explore which features are considered the most relevant by the ML models.

4.2 Results and Discussion

To begin our data analysis process and to train the models, we conducted experiments using varying releases per each project we have in the dataset. Initially, we used the code smells features to train the model. To generate the training models, the *Weka* tool takes as input an excel file that contains : the code smells data for all the projects and releases, one refactoring operation, and a “yes” or “no” value indicating whether upcoming release would undergo a the refactoring operation. Then we chose the classifier and we selected 10-cross validation as an option type and then pressed the start button to let the tool generate the results.

The 10-fold cross-validation is a technique used in machine learning to evaluate the performance of a classifier algorithm on a given dataset [9]. In this technique, the dataset is divided randomly into 10 equal-sized parts called "folds" The classifier is trained on 9 of the folds and tested on the remaining fold. This process is repeated 10 times (each fold serving as the test set once) and the average performance across all 10 iterations is calculated. The use of 10-fold cross-validation helps to reduce bias in the data or the selection of training/test sets, making it a reliable method for evaluating classifier performance.

We followed the same procedure for the 20 identified refactoring operations, repeating the experiments for each operation. Then, we merged the results into a single Excel file to calculate the average of all 20 refactoring operations.

We began the experiments with a dataset comprising 13 projects and one release per project. We then added another release per project and repeated the experiment until there were 5 releases per project. In addition, we applied the machine learning algorithms to the same dataset and exported the results for analysis.

Figure 2 summarizes our findings using the three scores explained in the previous section: precision, recall, and F-Measure, varying different releases per project. Also, it shows the results of the different machine learning algorithms.

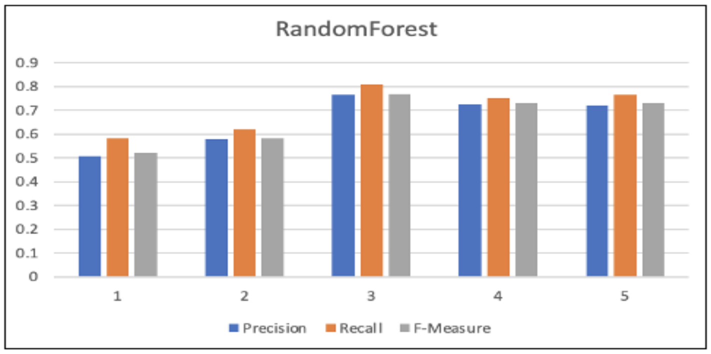
From the results, we can observe that starting from a dataset that contains projects with three releases, we can obtain good results. The precision, recall and f-measure scores of a dataset of projects with three releases (*scenario 1*) are far better than the dataset of projects with one release (*scenario 2*) and with two releases (*scenario 3*). *Scenario 2* has the lowest scores in all of the executed ML algorithms.

For instance, using the RandomForest algorithm, the average of precision, recall, and F-measure scores of the first scenario are 76%, 80%, 76% respectively, while those of the second scenario are only 50%, 58%, 52% respectively.

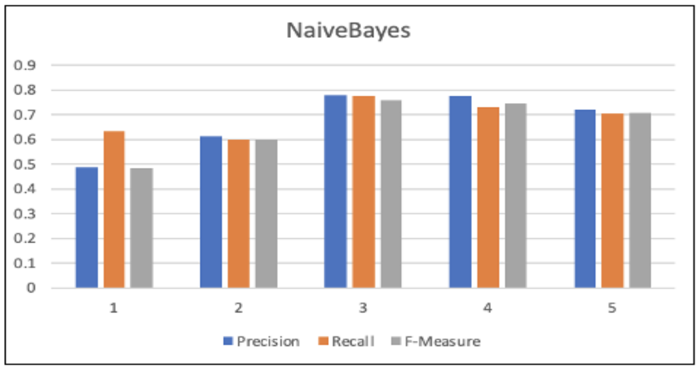
We realized that setting a refactoring operation to "no" in the entire training set affects the ability of the machine learning tool to comprehend and predict. Additionally, insufficient instances in the training set to perform classification lead to results that are mathematically undefined, denoted by a question mark.

Overall, based on our analysis of the three ML algorithms, we can conclude that data size significantly impacts the accuracy of results, and the results rely on the dataset's size. Therefore, we affirm that a dataset comprising at least three releases per project is necessary to achieve good results on average. This conclusion provides an answer to *RQ1*.

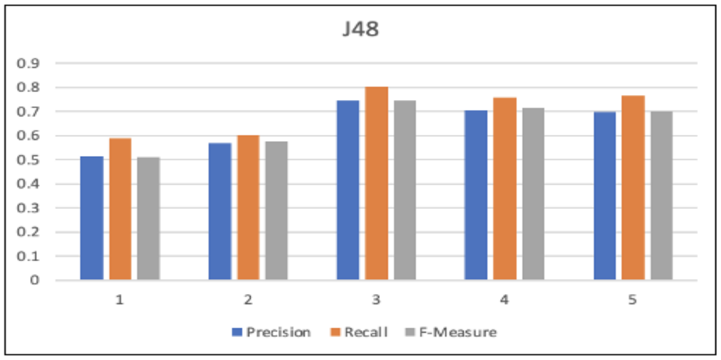
As described in Figure 2, we were able to train models that based on code smells on different projects with three releases where the refactoring operations were predicted with an average score higher than 77%. As for the comparison between the three ML algorithms, we can observe that the NaiveBayes algorithm shows better scores than the other algorithms using a dataset of projects with different releases and using the code smells features. For instance, NaiveBayes algorithm has 77% precision which is higher than the RandomForest algorithm with 76% precision and J48 algorithm with 74% precision.



(a) Random Forest Based on Code Smells



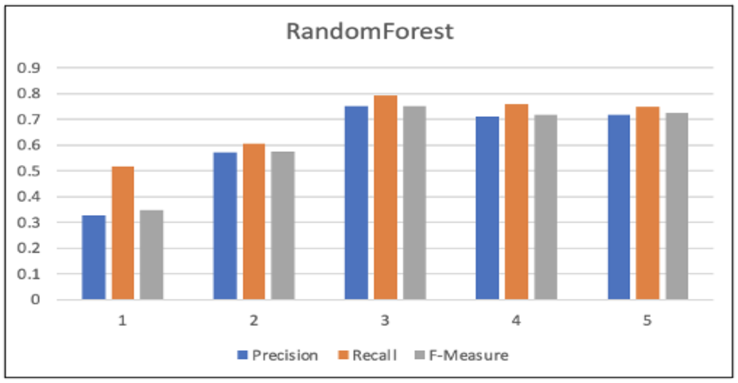
(b) Naive Bayes Based on Code Smells



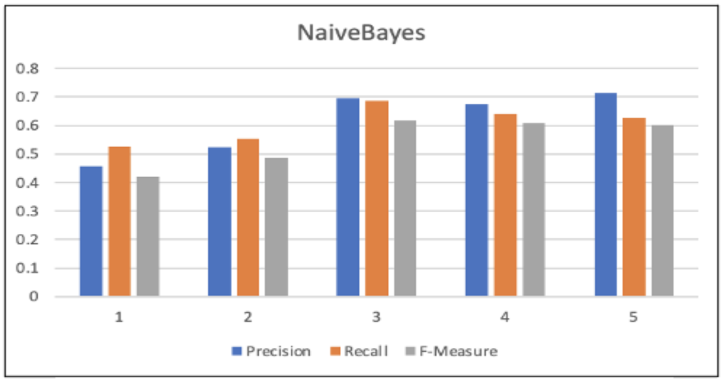
(c) J48 Based on Code Smells

Figure 2: The comparison of average results based on the value of code smells on different project releases.

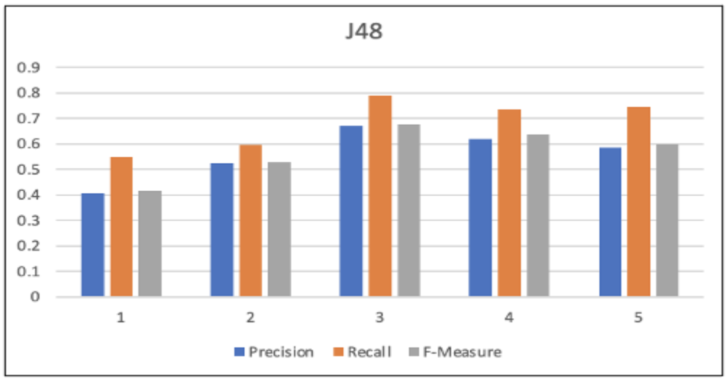
In this study, we repeated the same experiments but with the use of other features to train the prediction models since feature selection plays a critical role in the quality of the obtained models. This was done to explore the possibility of achieving higher scores. Figure 3 shows the results of using quality metrics features to train the prediction models.



(a) Random Forest Based on Quality Metrics



(b) Naive Bayes Based on Quality Metrics



(c) J48 Based on Quality Metrics

Figure 3: The comparison of average results based on the value of quality metrics on different project releases.

Similar to code smells, we observed that the highest results were achieved for projects with three releases, as depicted in Figure 3. However, there were differences in the performance of algorithms. For instance, RandomForest had the highest precision of 75%, outperforming NaiveBayes with 69% precision and J48 with 67% precision.

Our objective is to enhance the results obtained from our study. Upon comparing Figure 2 from the previous question, which displays the comparison of results obtained from code smells, with Figure 3, which compares results from quality metrics, we observed that code smells provide marginally better results than quality metrics. Notably, the difference between the two is not substantial; for instance, while the highest accuracy attained in Figure 2 is 77% with the NaiveBayes algorithm, in Figure 3, it is 75% with the RandomForest algorithm.

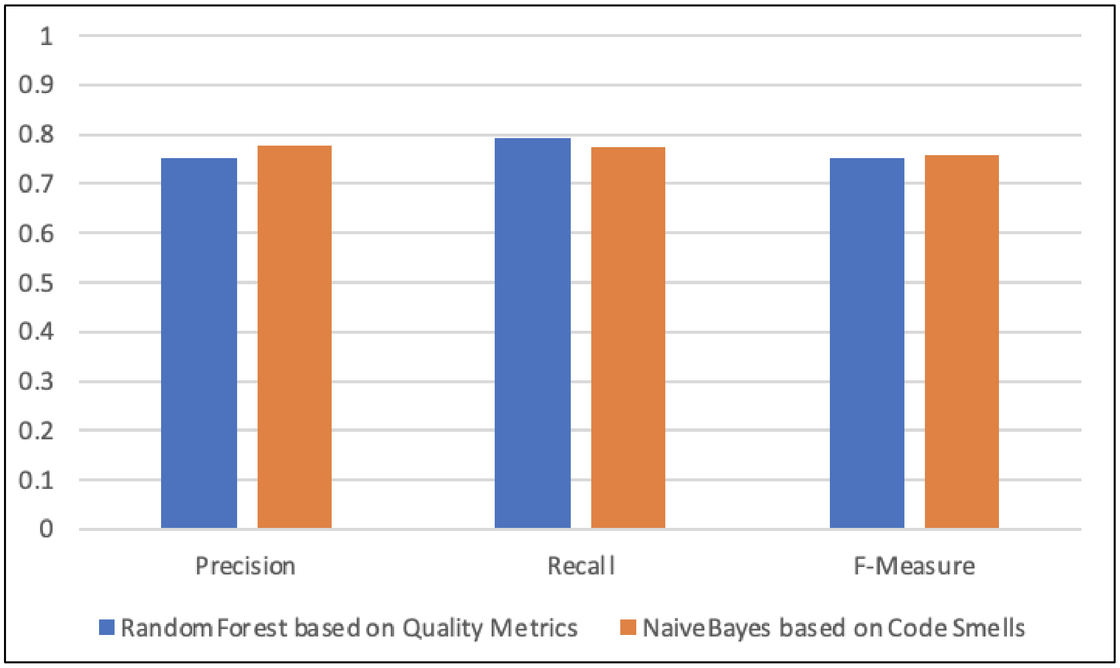


Figure 4: Comparison between the best of code smells and quality metrics average scores

The comparison of the best result obtained from code smells and quality metrics, with three releases, is displayed in Figure 4. We observed that the precision and F-Measure scores were higher when using code smells, whereas only recall was higher with quality metrics. Since precision is a crucial evaluation metric for our study, we can conclude that code smells features can be considered better than the quality metrics features.

After that, we conducted a testing process to validate the effectiveness of our trained models. Test sets were created for each refactoring operation that we will predict in the release that follows the releases in the training set. In each test set, the "yes" or "no" refactoring operations values were replaced with a question mark to allow the machine learning algorithm to predict them (the value "yes" indicates the presence of the refactoring operation in that release, while "no" indicates its absence).

The NaiveBayes classifier, which yielded the best scores (using the code smells features), was chosen for the prediction process. We applied these steps to predict the next project's release for all 20 refactoring operations across 13 projects, resulting in 260 test cases.

Next, we compared the predictions made by the *Weka* tool with the actual values obtained from *Refactoring Miner* tool, which was used to examine each release and identify any refactoring operations that had been performed. After manual inspection, we found that the tool accurately predicted 192 out of 260 cases, indicating a success rate of approximately 74%. This result confirms the reliability of our selected method for predicting future refactoring operations using code smells values.

Figure 5 shows in details the validation results, where the prediction was validated for each refactoring operation for all 13 projects.

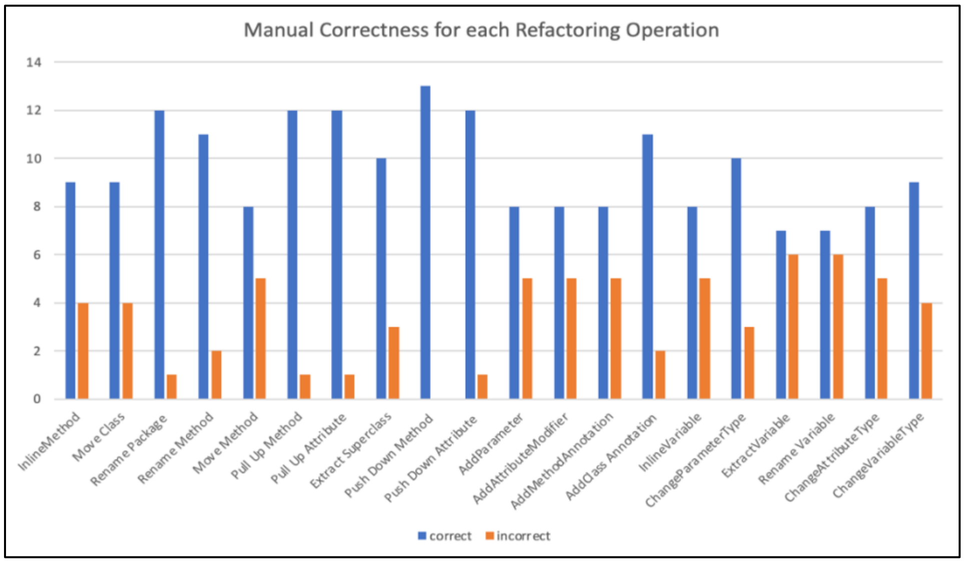


Figure 5: Manual Correctness for each Refactoring Operation for 13 projects

We performed a separate analysis of each project to identify correctly predicted refactoring operations and the incorrect ones. Figure 6 shows that the percentage of the correct prediction is higher than the percentage of the incorrect prediction in all projects, except for two projects, where the percentage was equal. For instance, in project 12, we were able to predict 19 out of 20 refactoring operations that should be applied in the next release.

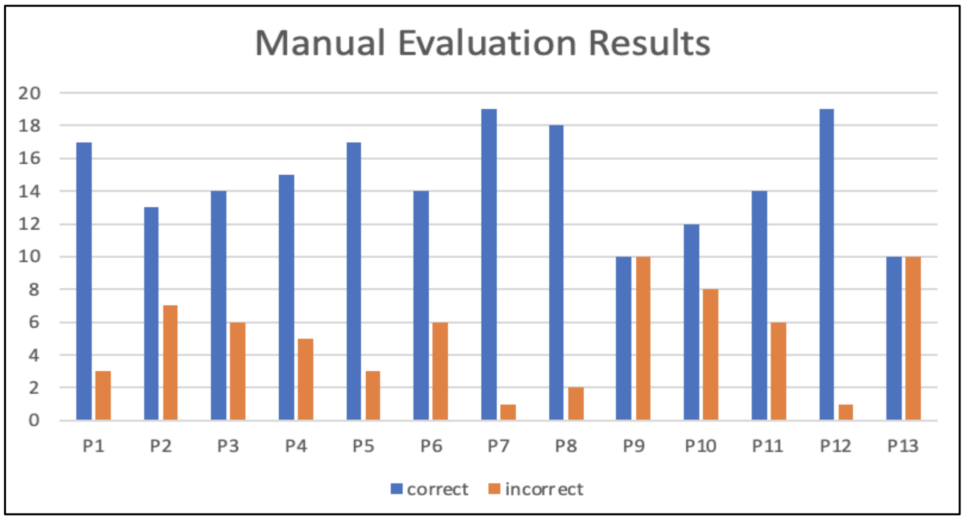


Figure 6: Manual Evaluation Results.

We anticipate several factors that may influence the outcome of refactoring operation predictions. One of these factors is the quality and maintenance needs of a project. Some releases may require a significant number of refactoring operations to address code smells and improve performance, while others may require only a few. Since our training sets are based on code smells, this should be considered, as the machine learning tool can predict the presence or absence of refactoring operations based on these values. Another factor that may affect the outcome is the selection of which refactoring operations to analyze and predict. Our study focused on 20 specific refactoring operations, but expanding the study to include additional operations or selecting different ones may impact the results positively or negatively. However, based on the verification results, we can conclude that it is possible to predict refactoring operations in a project's next release, which answers *RQ2.*

4.3 Threats to Validity

In this Section, we identify the limitations that may have an impact on our study and results.

First, our study was limited to open sources in Git repositories and we a anlyzed 13 projects. Adding more projects to the analysis may yield different results. We deliberately selected projects of varying sizes to increase the diversity of our sample, and the findings may differ if we limit the selection to a specific size range. Additionally, all the projects in our study were implemented in the Java programming language, and thus, we cannot generalize our results to other programming languages without further investigation.

Second, we analyzed only 20 refactoring operations identified by *Refactoring Miner*, which may be a threat to the validity of our results as it does not cover all refactoring categories described by Fowler et al. [3]. However, it is noteworthy that among these 20 operations, more than 10 are the most commonly used refactoring types [39] [40]. Our results may have been affected by the selection of the set of refactoring operations analyzed.

The third limitation of our study is that we focused on analyzing nine code smells prevalent in almost all versions of all selected projects. This was done to ensure that we had sufficient data for each code smell, thus avoiding the problem of zero values that could affect the accuracy of the machine learning model. However, it is important to note that choosing different code smells may impact our results.

In conclusion, software metrics are vital for improving software quality and developer productivity. For this reason, we focused on the CK quality metrics in our study since they have been shown to have a significant association with software defects according to previous research [31][32]. If we had selected a different set of metrics, our results would have differed from what we obtained using the CK metrics.

Chapter 5

Conclusion & Future Work

In this thesis, we investigated the effectiveness of using the history of code smells and quality metrics to predict refactoring operations in software development using machine learning and data mining algorithms. By analyzing various datasets and experimenting with different algorithms, the study demonstrated that predictive models could be generated, enabling developers to identify the type of refactoring operations to be applied in their project's next release.

As part of our investigation, we also conducted a manual evaluation to gauge the effectiveness of our approach using a test dataset. Our analysis revealed that we correctly predicted refactoring operations for most of the projects with an average accuracy higher than 77%. However, it is important to mention that the results depend on the quality and quantity of the used training set and the different extracted and applied features.

Several areas of future work can be explored in the context of refactoring operation prediction. One direction is to examine the integration of refactoring prediction models into existing development tools, such as integrated development environments (IDEs) to provide real-time feedback to developers. Another direction is to investigate the use of deep learning such as neural networks techniques to improve the accuracy of prediction models.

References

[1] Bass, L., Clements, P. & Kazman, R. (2003). “Software Architecture in Practice.” *Addison-Wesley*. ISBN: 9780321154958. https://www.amazon.com/Software-Architecture-Practice-2nd-Bass/dp/0321154959

[2] Watts, S & Kidd, C. (2018). "What is Code Refactoring? How Refactoring Resolves Technical Debt." *BMC Blogs*. Retrieved December 17, 2022. https://www.bmc.com/blogs/code-refactoring-explained/

[3] Fowler, M., Beck, K., Brant, J., Opdyke, W., & Roberts, D. (1999). "Refactoring: Improving the Design of Existing Code." *Addison-Wesley Longman Publishing Co. https://www.amazon.com/Refactoring-Improving-Design-Existing-Code/dp/0201485672v*

[4] Kaur, A. & Kaur, M. (2016). “Analysis of Code Refactoring Impact on Software Quality.” *MATEC Web of Conferences*. 57. 02012. https://www.researchgate.net/publication/302977540_Analysis_of_Code_Refactoring_Impact_on_Software_Quality

[5] Paramshetti, P. & Phalke, P. (2014). "Survey on Software Defect Prediction Using Machine Learning Techniques," *International Journal of Science and Research*. https://www.academia.edu/66176405/Survey_on_Software_Defect_Prediction_Using_Machine_Learning_Techniques

[6] Tsantalis, N., Ketkar, A. & Dig, D. (2020). "RefactoringMiner 2.0." *IEEE Transactions on Software Engineering*. 1-1. https://ieeexplore.ieee.org/document/9136878

[7] Silva, D., Tsantalis, N. & Valente, M. (2016). "Why We Refactor? Confessions of GitHub Contributors." *Proceedings of 24th International Symposium on the Foundations of Software Engineering.* 1-12. https://dl.acm.org/doi/10.1145/2950290.2950305

[8] Brown, W., Malveau, R., Mccormick, H. & Mowbray, T. (1998). “AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis.” *New York: John Wiley and Sons*. ISBN: 9780471197133 https://www.researchgate.net/publication/245429350_AntiPatterns_Refactoring_Software_Architectures_and_Projects_in_Crisis

[9] Trevor H., Robert T. & Jerome F. (2009). “The Elements of Statistical Learning: Data Mining, Inference, and Prediction, Second Edition.” *Springer*. https://link.springer.com/book/10.1007/978-0-387-84858-7

[10] Moha, N., Guéhéneuc, Y., Duchien, L. & Meur, A. (2010). “DECOR: A Method for the Specification and Detection of Code and Design Smells.” *IEEE Transactions on Software Engineering*. 36. 20-36. https://www.researchgate.net/publication/232650234_DECOR_A_Method_for_the_Specification_and_Detection_of_Code_and_Design_Smells

[11] Ratzinger, J., Sigmund, T., Vorburger, P. & Gall, H. (2007). “Mining Software Evolution to Predict Refactoring”. *Proceedings of 1st International Symposium on Empirical Software Engineering and Measurement.* 354-363. https://ieeexplore.ieee.org/document/4343763

[12] Sagar, P., AlOmar, E., Mkaouer, M., Ouni, A., & Newman, C. (2021). “Comparing Commit Messages and Source Code Metrics for the Prediction Refactoring Activities.” *Algorithms*. 14.289. https://www.researchgate.net/publication/355015027_Comparing_Commit_Messages_and_Source_Code_Metrics_for_the_Prediction_Refactoring_Activities?_sg=iCQhtkYVOYSWyTPlehkZgM4EzjhiZy9fp-UdYeg9fWtB7NbkGiGM8Ev8PH07SuGqH83FajEvX7t-eKM

[13] Sjoberg, D. I. K., Yamashita, A., Anda, B. C. D., Mockus, A., & Dyba, T. (2013). “Quantifying the Effect of code smells on Maintenance Effort.” *IEEE Transactions on Software Engineering*. 39(8), 1144–1156. https://www.researchgate.net/publication/260648949_Quantifying_the_Effect_of_Code_Smells_on_Maintenance_Effort

[14] Deligiannis, I., Stamelos, I., Angelis, L., Roumeliotis, M. & Shepperd, M. (2004). "A controlled experiment investigation of an object-oriented design heuristic for maintainability." *Journal of Systems and Software*. 72. 129-143. https://doi.org/10.1016/S0164-1212(03)00240-1

[15] Olbrich, S., Cruzes, D. & Sjøberg, Dag. (2010). "Are all code smells harmful? A study of God Classes and Brain Classes in the evolution of three open source systems." *IEEE International Conference on Software Maintenance, ICSM*. 1-10. https://ieeexplore.ieee.org/document/5609564

[16] Zhang, M., Hall, T. & Baddoo, N. (2011). "Code Bad Smells: A review of current knowledge." *Journal of Software Maintenance*. 23. 179-202. https://www.researchgate.net/publication/220673981_Code_Bad_Smells_A_review_of_current_knowledge

[17] Cedrim, D., Garcia, A., Mongiovi, M., Gheyi, R., Sousa, L., de Mello, R., Fonseca, B., Ribeiro, M. & Chávez, A. (2017). "Understanding the impact of refactoring on smells: a longitudinal study of 23 software projects." *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 465-475. https://doi.org/10.1145/3106237.3106259

[18] Al Dallal, J. (2014). "Identifying Refactoring Opportunities in Object-Oriented Code: A Systematic Literature Review." *Information and Software Technology*. 58. https://www.researchgate.net/publication/264791334_Identifying_Refactoring_Opportunities_in_Object-Oriented_Code_A_Systematic_Literature_Review

[19] Sharma, T., Mishra, P. & Tiwari, R. (2016). "Designite: a software design quality assessment tool." *Proceedings of 1st International Workshop*. 1-4. https://dl.acm.org/doi/10.1145/2896935.2896938

[20] Dragomir, M. (2015). "Understand." *Softpedia*. Retrieved January 17, 2023, from https://www.softpedia.com/get/Programming/Coding-languages-Compilers/Understand.shtml

[21] Santos, J., Rocha-Junior, J., Prates, L., Nascimento, R., Freitas, M. & Mendonça, M. (2018). "A systematic review on the code smell effect." *Journal of Systems and Software*. 144. 450-477. https://www.researchgate.net/publication/327993118_A_systematic_review_on_the_code_smell_effect

[22] Hamdi, O., Ouni, A., Alomar, E., Ó Cinnéide, M. & Mkaouer, M. (2021). "An Empirical Study on the Impact of Refactoring on quality metrics in Android Applications." *2021 IEEE/ACM 8th International Conference on Mobile Software Engineering and Systems (MobileSoft)* . 28-39. https://ieeexplore.ieee.org/document/9460932

[23] Halstead, M. (1977). "Elements of Software Science (Operating and Programming Systems Series)." *Elsevier Science Inc*. New York, NY, USA. https://dl.acm.org/doi/10.5555/540137

[24] Ganjare, S., Kulkarni, K., Jadhav, Y. & Hanchate, D. (2015). "Measuring Structural Code Quality Using Metrics." *Inventi Rapid: Soft Engineering*. https://www.researchgate.net/publication/278786599_Measuring_Structural_Code_Quality_Using_Metrics

[25] Bouckaert, R., Frank, E., Hall, M., Kirkby, R., Reutemann, P., Seewald,A. & Scuse, D. (2016). "WEKA Manual for Version 3-8-1." *Hamilton, New Zealand: University of Waikato.* https://sci2s.ugr.es/sites/default/files/files/Teaching/GraduatesCourses/InteligenciaDeNegocio/Curso19-20/WekaManual-3-8-1.pdf

[26] McCabe, T. J. (1976). "A Complexity Measure." *IEEE Transactions on Software Engineering*. vol. 2, no. 4, pp. 308-320, December 1976.

[27] Kochhar, P., Thung, F., & Lo, D. (2014). "Automatic Fine-Grained Issue Report Reclassification." *Proceedings of the IEEE International Conference on Engineering of Complex Computer Systems, ICECCS.* https://ieeexplore.ieee.org/document/6923127

[28] Chahal, H. (2018). "Comprehensive Analysis of Data Mining Classifiers using WEKA." *International Journal of Advanced Computer Research.* https://www.researchgate.net/publication/325605143_Comprehensive_Analysis_of_Data_Mining_Classifiers_using_WEKA

[29] Patil, T. & Sherekar, S. (2013). "Performance Analysis of Naive Bayes and J48 Classification Algorithm for Data Classification." *International Journal of Computer Science and Applications.* https://www.researchgate.net/publication/305155554_Performance_Analysis_of_Naive_Bayes_and_J48_Classification_Algorithm_for_Data_Classification

[30] Fernandes, E., Oliveira, J., Vale, G., Paiva, T. & Figueiredo, E. (2016). “A Review-based Comparative Study of Bad Smell Detection Tools.” *20th International Conference on Evaluation and Assessment in Software Engineering (EASE).* 1-12. https://www.researchgate.net/publication/303515517_A_Review-based_Comparative_Study_of_Bad_Smell_Detection_Tools

[31] Chidamber, S.& Kemerer, C. (1994). "A Metrics Suite for Object Oriented Design." *IEEE Transactions on Software Engineering.* 476–493. https://doi.org/10.1109/32.295895

[32] Subramanyam, R. & Krishnan, M. (2003). "Empirical Analysis of CK Metrics for Object-Oriented Design Complexity: Implications for Software Defects." *IEEE Transactions on Software Engineering.* 297- 310. https://doi.org/10.1109/TSE.2003.1191795

[33] Vujovic, Z. (2022). "CASE REPORT A Case Study of the Application of WEKA Software to Solve the Problem of Liver Inflam- mation." *Archives of Clinical and Experimental Surgery (ACES).* 10. 01-13. https://www.researchgate.net/publication/357747772_CASE_REPORT_A_Case_Study_of_the_Application_of_WEKA_Software_to_Solve_the_Problem_of_Liver_Inflam-_mation

[34] Alomar, E., Mkaouer, W. & Ouni, A. & Kessentini, M. (2019). “On the Impact of Refactoring on the Relationship between Quality Attributes and Design Metrics.” *ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM).* 1-11. https://www.researchgate.net/publication/336630150_On_the_Impact_of_Refactoring_on_the_Relationship_between_Quality_Attributes_and_Design_Metrics

[35] Kumar, L. & Satapathy, S. & Murthy, L. (2019). “Method Level Refactoring Prediction on Five Open Source Java Projects using Machine Learning Techniques.” *The 12th Innovations*. 1-10. https://www.researchgate.net/publication/330972151_Method_Level_Refactoring_Prediction_on_Five_Open_Source_Java_Projects_using_Machine_Learning_Techniques

[36] Aniche, M., Maziero, E. & Durelli, R. & Durelli, V. (2020). “The Effectiveness of Supervised Machine Learning Algorithms in Predicting Software Refactoring.” *IEEE Transactions on Software Engineering*. PP. 1-1. https://www.researchgate.net/publication/344562378_The_Effectiveness_of_Supervised_Machine_Learning_Algorithms_in_Predicting_Software_Refactoring

[37] Akour, M., Alenezi, M. & Alsghaier, H. (2022). “Software Refactoring Prediction Using SVM and Optimization Algorithms.” *Processes.* https://www.researchgate.net/publication/362694597_Software_Refactoring_Prediction_Using_SVM_and_Optimization_Algorithms

[38] Tan, L., & Bockisch, C. (2019). “A Survey of Refactoring Detection Tools.” *Software Engineering.* https://www.semanticscholar.org/paper/A-Survey-of-Refactoring-Detection-Tools-Tan-Bockisch/045c709957f82e06b22328a2dea15a1b71bf0bf4

[39] Golubev, Y., Kurbatova, Z., AlOmar, E., Bryksin, T. & Mkaouer, W. (2021). “One thousand and one stories: A large-scale survey of software refactoring.” *In Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (pp. 1303-1313).* https://www.researchgate.net/publication/353284653_One_Thousand_and_One_Stories_A_Large-Scale_Survey_of_Software_Refactoring

[40] Murphy-Hill, E., Parnin, C. & Black, A. (2009). “How we refactor, and how we know it.” *IEEE 31st International Conference on Software Engineering.* 287-297. https://ieeexplore.ieee.org/document/5070529