FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Mining Software Repositories to Improve Refactoring Assistants

Diogo Faria

Mestrado em Engenharia Informática e Computação

Supervisor: Prof. Ademar Aguiar

Second Supervisor: Prof. Sara Fernandes

July 26, 2024

Mining Software Repositories to Improve Refactoring Assistants

Diogo Faria

Mestrado em Engenharia Informática e Computação

Approved in oral examination by the committee:

July 26, 2024

Resumo

O desenvolvimento de software sofre uma evolução constante, aumentando a sua complexidade ao longo do tempo, sendo que certas necessidades começam a aparecer. Dentro delas, algumas das mais importantes podem ser a necessidade de código eficiente e sustentável.

Refatoração, como o processo de alterar um sistema de software de uma forma que não altere o comportamento externo do código, mas melhore a sua estrutura interna, tem um papel em tratar de algumas destas necessidades. No entanto, abordagens tradicionais de refatoração podem ser vistas como complicadas, tediosas e demoradas.

Tradicionalmente, refatoração requere que um desenvolvedor identifique oportunidades e depois decida que refatoração deverá fazer. Depois de ser feita, ainda precisa de verificar que a refatoração preservou o comportamento total do sistema.

No contexto de um *plugin* de recomendação de refatorações, LiveRef, nós exploramos uma abordagem que utiliza aprendizagem computacional para melhorar o processo de identificação e sugestão de refatorações, particularmente no caso de *Extract Method* e *Extract Class*.

Neste *plugin*, nós trocamos o método convencional baseado em limites com um modelo de classificação dinâmico, permitindo-o aprender continuamente enquanto está a ser utilizado e adaptar-se a novos contextos de software.

Nós aproveitamos o poder de mineração de dados em repositórios de software existentes para extrair dados da vida real, criando um conjunto de dados rico e variado que melhora o treino do modelo e garante que se aplica a diversos cenários no mundo real.

Para validar o que desenvolvemos, uma análise automática foi realizada de forma a poder comparar o número de sugestões que o plugin recomenda, usando tanto a versão original do plugin e a versão depois das nossas mudanças, com dados reais de desenvolvedores de software.

Nós acreditamos que o uso de dados reais é crucial para o processo de sugestões de refatorações, pois uma refatoração de livro didático não é necessariamente o que desenvolvedores vão realizar nas suas operações dia-a-dia. Assim, uma ferramenta de recomendação de refatorações que consegue ter em conta uma grande variedade de contextos enquanto se adapta ao seu utilizador ao longo do tempo vai aumentar a sua utilização pelo desenvolvedor, levando a sistemas de software mais bem preservados.

Abstract

Software development faces an ever-constant evolution, increasing its complexity over time, and with it, particular needs start to arise. Among them, some of the most important can be considered the need for efficient and maintainable code.

Refactoring, as “the process of changing a software system in a way that does not alter the external behaviour of the code yet improves its internal structure", plays a role in addressing some of these needs. However, traditional approaches to refactoring can be perceived as cumbersome, tedious, and time-consuming.

Traditionally, refactoring requires a developer to identify opportunities and then decide which refactoring they should apply. After it is performed, they still need to ensure that the refactoring preserves the overall system behaviour.

In the context of an existing refactoring recommendation plugin, LiveRef, we explored an approach that uses machine learning to improve the identification and suggestion process, particularly on the Extract Method and Extract Class refactorings.

Within this plugin, we replace the conventional threshold-based method with a dynamic classification model, allowing it to continuously learn as it is being used and adapt to new software contexts.

We leveraged the power of data mining in existing software repositories to extract real-life data, creating a rich and varied dataset that enhances the model’s training and ensures that it applies to diverse real-world scenarios.

To validate what we developed, an automated analysis was performed in order to be able to compare the number of suggestions the plugin recommended, using both the original plugin version and the one after our changes, with real-life developer refactoring data.

We believe that the use of real-life data is crucial in the refactoring suggestions process, as a textbook refactoring isn’t necessarily what developers are going to perform in their day-to-day operations. Thus a refactoring recommendation tool that is able to take into account a larger variety of contexts while adapting to its user over time will increase a developer’s use of it, leading to better-maintained software systems.

Acknowledgements

First, I would like to thank Ademar Aguiar, who was able to help and guide me throughout this entire work, providing me with guidance and support until the very end. I would also like to extend my thanks to Sara Fernandes, who provided me with the tools needed to complete this endeavour.

My friends, both in and out of university, who've been there for me every step along the way, even if they didn't necessarily understand exactly what I'm doing, I'm not going to forget your help. Thank you. Some of you I've been close for years, others only more recently, but that doesn't change how much you've meant to me and your unwavering support, quick helping hand whenever needed, or just simply offering an avenue to relax and gather my thoughts was crucial to get me where I am today. Long live all the magic we made.

I want to thank my family, who's support hasn't faded for a single moment in my life and who I know are as excited as me, or even more, for me to take the next step in life, to lead my own way. If it wasn't for your unwavering support, I wouldn't have turned out the way I did, and none of my accomplishments would have ever been my own.

Lastly, I want to thank my grandmother, who definitely didn't understand what I do, but would never forget to ask if I was okay, who I would've loved to see as I complete this major step in life, but who missed it by mere moments. You're alive in my head. She's someone whose leaving never felt possible, who I took for granted but who played a larger role in my life than I sometimes realise. If I didn't know better, I'd think you were still around.

Diogo Faria

“Hold on to the memories, they will hold on to you.”

Taylor Swift

Contents

List of Figures

List of Tables

Abbreviations

AST	Abstract Syntax Tree
LCOM	Lack of Cohesion in Methods
MSR	Mining Software Repositories
SAR	Self Affirmed Refactoring
SVMs	Support Vector Machines

Chapter 1

Introduction

As software projects move forward in their life cycle, it is inevitable that their complexity keeps increasing. The introduction of different software developers, programming styles, and even naming conventions, or budget and time constraints that force rushed designs and implementations all lead to lower code quality and a harder-to-comprehend code corpus that will only result in repercussions for any future development.

1.1 Context

In order to deal with these issues, refactoring is introduced as a process to improve the code's structure, such as its readability and maintainability, without altering its original behaviour [Fow18]. The concept of refactoring is not new, though as time goes on and software becomes increasingly complex, the need for refactoring keeps increasing.

Traditional refactoring operations tend to follow certain steps, from the identification of specific issues to the selection of what refactoring is the most suitable and, finally, to performing it. These steps are often guided by well-established principles and patterns, allowing developers to improve code quality systemically.

1.2 Motivation

These steps, however, rely heavily on manual inspection and human intuition, thus possibly resulting in the introduction of errors in the code while also leading to a perception that it is a tedious and time-consuming process.

Aside from that, as they undergo this operation, developers often find themselves spending quite some time searching through a refactoring catalogue filled with various well-researched and -documented refactoring patterns to understand how to fix their particular issue.

Several solutions have arisen as an answer to some of these issues, namely refactoring recommendation tools, which seek to automate the refactoring process, automated testing, to reduce the propensity for errors, and refactoring metrics, to help the developer find where refactoring is needed, though they, unfortunately, are unable to improve on everything.

1.3 Problem

Approaches and tools to improve the refactoring effectiveness and efficiency taken by developers often use code quality metrics that need to be calculated and categorized via thresholds for each refactoring type they support. [MT04]

However, these thresholds aren't necessarily the best because they are created using older, sometimes biased, and unreliable data and may lead to missed or inappropriate refactoring suggestions, especially in complex and ever-evolving environments.

Aside from that, suggestions born from threshold-based methods are subjective to the authors who created them, not to the users who will eventually make use of them, possibly leading to recommendations that don't align with their users.

Ultimately, the main problem is that these threshold-based suggestions remain stuck at their moment of creation and are incapable of adapting to new and more updated contexts.

1.4 Objectives

Thus, the goal is that by tapping into the knowledge found in software repositories, we can obtain data from real-life instances of refactorings, meaning change data found in software repositories where certain refactorings have occurred. This real-life data can then be used in the development of models that can capture the complex relationships between metrics to provide refactoring suggestions.

As such, we intend to mine software repositories to extract instances of refactoring operations, saving a set of code quality metrics from the change versions of the refactorings into a knowledge base.

With this knowledge base, we want to understand and compare how threshold-based method refactoring suggestions fare against real-life data, allowing us to understand its limitations.

Furthermore, we want to explore an alternative to the threshold method in the form of classification models that feed on the knowledge base to provide the refactoring suggestions and are able to continuously adapt as they're being used to be better suited towards their users.

1.5 Document Structure

Aside from the introduction, this dissertation contains 5 more chapters. In Chapter chapter 2 the state of the art in the key topics required for this thesis is described, along with small background knowledge that may be required. In Chapter 3 the problem is presented, while its solution is described in Chapter 4, and its validation process and results are detailed in Chapter 5. Lastly, the conclusion and future work are in Chapter 6.

Chapter 2

State of the Art

As mentioned before, the objective of this thesis is to develop an automated data mining approach that would be able to identify refactoring opportunities in software repositories based on code quality metrics and suggest how to perform such refactorings, which requires a deep understanding of topics such as refactoring, data mining, mining software repositories (MSR), and code quality metrics.

With this in mind, we decided to perform a literature review focusing on these three topics, aiming to understand the current state-of-the-art and possible open issues.

2.1 Research Strategy

We performed a quasi-literature review, taking into account the steps required to perform one, though not following them to the letter. We began by creating the research questions that would guide us throughout the rest of this review. Based on them, we created research queries to allow us to gather any relevant data to our purposes, which was then followed by a review of said data to understand its content. Lastly, we summarized our obtained insights.

2.1.1 Research Questions

Our objective is to use real-life refactoring data present in software repositories to build a classification model in order to provide refactoring recommendations, believing it may prove itself better than the conventional methods. Thus, we required insights on mining software repositories to understand the type of data present in such repositories and how it's obtained, and if research on finding refactoring data in software repositories has been performed before and, if so, how was the data obtained. Aside from that, we wished to understand the current landscape of refactoring recommendation tools.

As a result, we formulated the following research questions:

RQ1 "Has refactoring identification using data mining techniques already been done? If so, how?"

RQ2 "What are the current techniques that can be used for automated identification of refactorings using data mining techniques on software repositories?"

RQ3 "How do refactoring recommendation tools work?"

2.1.2 Data Search and Retrieval

We carried out a search and selection process to gather literature, which was limited to the year 2023 when we conducted and wrote this review, focusing on the most recent papers as they would provide us with the most relevant information. To gather the research material, we used 4 digital libraries: Web of Science, Scopus, ACM and IEEE Xplorer.

The search queries we performed were the same for each library:

SQ1 ("data mining" OR mining) AND ("software repositories" OR repositories) AND refactoring

SQ2 (refactoring AND (recommendation OR suggestion OR prediction) AND (tool OR approach OR method))

Even though we used the same query for the different libraries, the location where they were searched within differs, with them being searched within the Article title, the Abstract, and the Keywords in Scopus and Web of Science, with the addition of them being searched in Keywords Plus in Web of Science, which are generated keywords by the library. IEEE Xplore had the query searched on all metadata, while for the ACM digital library, the search was only conducted for the Abstract as a search within more content led to over 2000 results for the first query and over 14000 for the second.

As can be seen in fig. 2.1, we collected the data from the different libraries for the 2 separate queries, which resulted in a large number of collected data for the second query. That led us to select only the top 20 results from each library, ordered by their relevance so as not to be overwhelmed. Having found the literature, we used our inclusion criteria defined in table 2.1 to refine our choices further.

Following this, we used the Snowball Method to acquire extra literature that we believed was relevant and would enhance our work efforts, thus leading us to have 26 research papers.

Number	SQ1 Criteria	SQ2 Criteria
1	Refactoring as a Main Topic	
2	Uses Software Repositories	References Refactoring Tools/Methods
3	Details MSR or Refactoring Activity Detection Method	Details the Refactoring Tool/Method
4	Provides Empirical Results or Case Studies	

Table 2.1: Inclusion Criteria

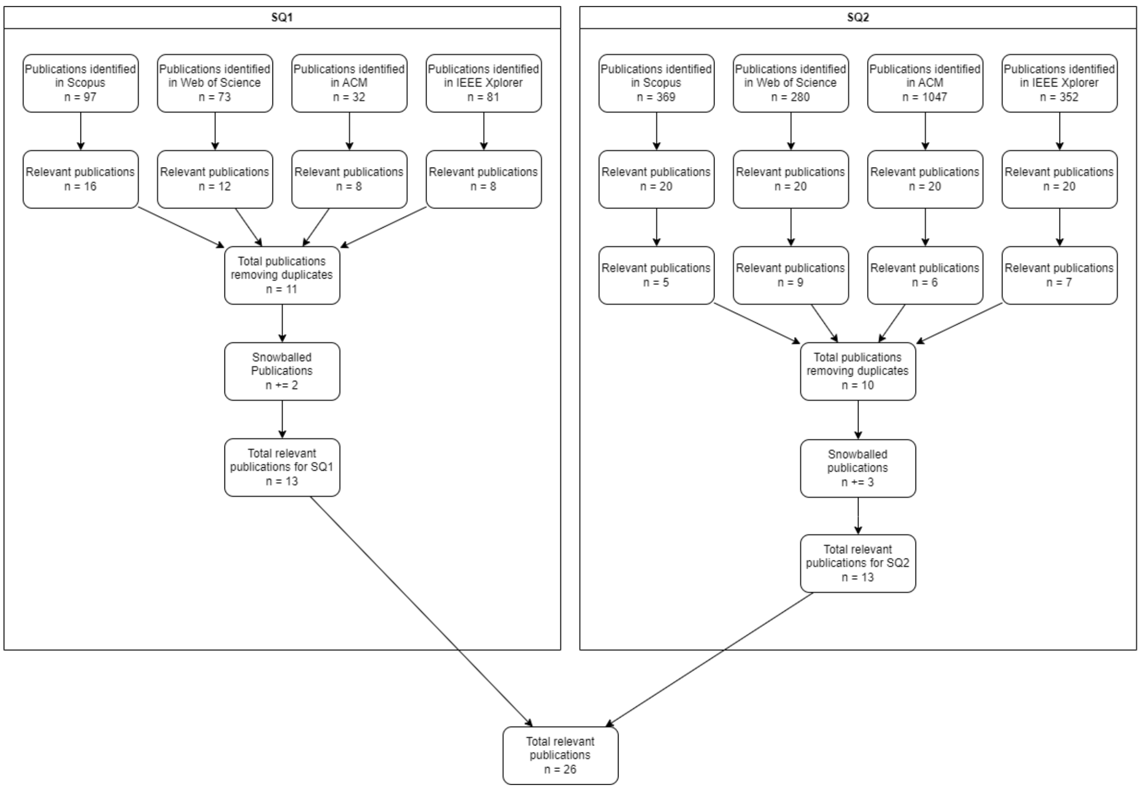

SQ1
Publications identified in Scopus n = 97
Publications identified in Web of Science n = 73
Publications identified in ACM n = 32
Publications identified in IEEE Xplorer n = 81
Relevant publications n = 16
Relevant publications n = 12
Relevant publications n = 8
Relevant publications n = 8
Total publications removing duplicates n = 11
Snowballed Publications n += 2
Total relevant publications for SQ1 n = 13
SQ2
Publications identified in Scopus n = 369
Publications identified in Web of Science n = 280
Publications identified in ACM n = 1047
Publications identified in IEEE Xplorer n = 352
Relevant publications n = 20
Relevant publications n = 20
Relevant publications n = 20
Relevant publications n = 20
Relevant publications n = 5
Relevant publications n = 9
Relevant publications n = 6
Relevant publications n = 7
Total publications removing duplicates n = 10
Snowballed publications n += 3
Total relevant publications for SQ2 n = 13
Total relevant publications n = 26

Figure 2.1: Literature Search and Selection Overview

The literature collection we compiled based on our inclusion criteria was the basis for our state-of-the-art research. However, aside from that collection, there was still relevant literature in our search results, namely regarding background knowledge about the various subject matters we were interested in.

Compiling that literature with the extra we acquired via either the Snowball method on the existing research papers or by directly searching for it using relevant keywords, we collected 20 research papers that would be used for the background section, leading us to a final count of 46 research papers.

2.2 Background

In order to understand the state-of-the-art regarding refactoring, how to identify and suggest it using data mining, and mining software repositories, one needs first to understand the basics that make these topics.

2.2.1 Refactoring

Over time, software systems increase in complexity and become more challenging to maintain, which may lead to a decline in code quality over time. Refactoring can be seen as an answer to this problem, as a process that consists of altering code and improving it in some way without its external behaviour changing [Fow18]. It has been shown to lead to faster programming by improving the software's design, an improvement in code readability [Pia], a decrease in software defects introduced after refactorings have been performed [FFYI] [RSG08], and better bug detection [Fow18].

Fowler [Fow18] described around 70 refactorings, a few of which are presented below:

- **Combine Functions into Class**: Whenever there is a group of functions that operates on common data, they can be grouped together into a single class containing them all;
- **Extract Function**: Whenever there is a fragment of code that requires time spent to understand what it means, then it should be extracted to its own function named function after the meaning of the code fragment;
- **Extract Variable**: Whenever there is a complex expression, meaning it is hard to read, it should be split into parts, creating new variables that would allow it to be more understandable;
- **Inline Function**: If a code is using too much indirection, meaning it keeps redirecting to other functions unnecessarily, it would be easier to inline them all into a simple one;
- **Introduce Parameter Object**: Groups of data that are regularly used together could, instead, be replaced by a single data structure, making it easier to understand the relationship between the data;
- **Rename Method**: If a function has a wrong name or non-self-explanatory one, then it should be renamed;
- **Rename Variable**: Whenever a variable name isn't self-explanatory, it should be renamed.

Deciding when and what to refactor is just as important as the refactoring itself [Fow18]. Thus, code smells were introduced as "a surface indication that usually corresponds to a deeper problem in the system" [Fow], and they have since been used as a sign of a need for refactoring.

Code smells, though, aren't always easy to find, with a study having found that they are not commonly identified during code reviews [HTL+22], which may lead to further defects down the line.

Knowing what to refactor isn't enough, as knowing how to do it properly is crucial. One study found that the percentage of faults likely induced by refactorings is low, around 15% [BCL+], though another found that 54% of bug-inducing commits present in the repositories they analysed contained at least one refactoring operation [BH], leading them both to recommend that more extensive code inspection, validation and verification are required when performing refactorings.

Furthermore, to effectively manage and improve large-scale software projects, it is essential to leverage data-driven insights. By analysing historical data in software repositories, developers can uncover patterns and gain a better understanding of how to perform several decisions, including refactoring.

2.2.2 Mining Software Repositories

Software repositories can be extensive, containing much information regarding historical data and have been traditionally used for archival purposes, and the mining software repositories field focuses on taking advantage of this information to improve traditional software [Has].

That information is divided into several different categories that people can choose to extract or not, the most prevalent being commit data and source code [SHPCP21]. This can be used for several purposes and with a range of focuses, such as software evolution, maintainability, development effort, and refactoring, among many others [FNJ+16], as shown in fig. 2.2.

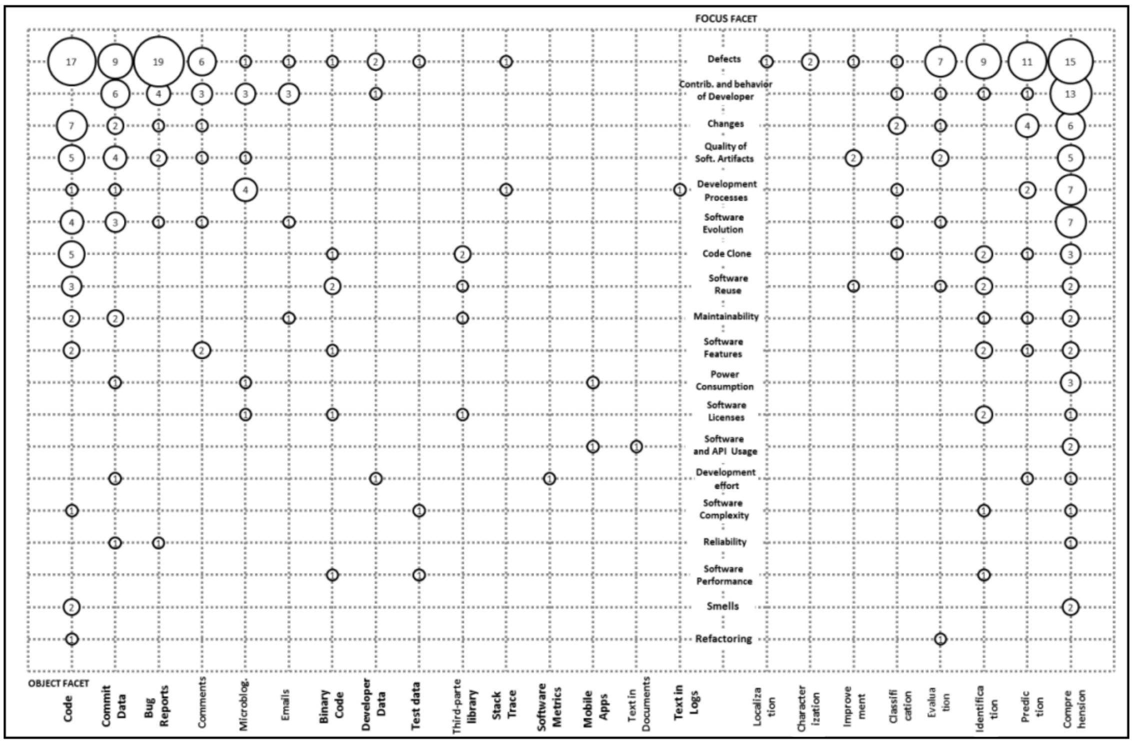


Figure 2.2: Focus x Object of analysis x Purpose of MSR studies (Extracted from [SHPCP21])

2.2.3 Data Mining

Han et al. [HPT22] define data mining as “the process of discovering interesting patterns, models, and other kinds of knowledge in large data sets”. It can be considered as extracting knowledge from data through various different means. In this section, we’ll present and explain some data mining concepts.

Text mining, as defined by Han et al. [HPT22], is about extracting information from text in the form of structures, patterns, and summaries. It’s usually characterized by processes such as the extraction of relations between entities, sentiment analysis, and text categorization, among others. Text categorization, for example, aims to assign analysed texts into predefined categories.

Frequent itemset mining pertains, as the name suggests, to the mining of frequent itemsets, which are sets of something that appear at least a certain number of predetermined times. An itemset can be considered closed in a certain data set if no other superitemset exists where this itemset appears with the same frequency. Frequent itemsets could also be considered closed or maximal depending on certain conditions. [HPT22]

A widely used data analysis is classification, which extracts models describing important data classes [HPT22]. Several classification techniques will be described further:

- **Bayes Classification**: Statistic classifiers, which predict the probability of the input being of a certain class, often exhibiting high accuracy and speed in large databases. [HPT22]
- **Perceptron**: Linear classifier, which can be modified for a classification task.
- **Logistic Regression**: A binary classification algorithm, based on a sigmoid function which maps the output of a linear regression model to a number between 0 and 1, therefore allowing the result to express how confident the model is with its own prediction. [HPT22]
- **Decision Tree**: It's a flowchart-like tree structure, where the leaf nodes represent classes and the non-leaf nodes represent a test on an attribute. Decision trees are simple to build, fast and can have great success. [HPT22]
- **Neural Networks**: They're a set of connected input-output units, which, depending on how these units are organized, result in different kinds of neural networks. For example, Perceptron and logistic regression can be viewed as a unit. [HPT22]

Chandola et al. [CBK09] define anomaly detection as "the problem of finding patterns in data that do not conform to expected behaviour", where these patterns can then be referred to as anomalies.

There exist several anomaly detection techniques, including classification-based ones, which can be categorized into two categories: multi-class and one-class classification techniques. As the names suggest, multi-class classification deals with training data which contains multiple normal classes, while the entirety of the training data used by one-class classification algorithms belongs only to a single class. [CBK09]

A few anomaly detection-based classification techniques are:

- **One-Class Support Vector Machines (SVMs)**: This technique learns a boundary that encompasses the training data instances, using kernels such as the radial basis function to capture complex regions. During testing, an instance is declared as normal if it falls within the learned region, otherwise, it is classified as anomalous. [CBK09]
- **Isolation Forest**: This method builds an ensemble of isolation trees for a given dataset, in which the instances that have short average paths are classified as anomalous. [LTZ]
- **Elliptic Envelope**: This models the data as a high dimensional Gaussian distribution with possible covariance between feature dimensions, meaning it tries to find a boundary ellipse

which contains most of the data and anything outside of this ellipse is classified as anomalous. [HRP+15]

2.2.4 Code Quality Metrics

A code quality metric can be described as a standard measure of which a software system or process possesses certain properties, which can help developers estimate the quality of their software [SA20].

Halstead [Hal77] has defined a number of metrics based on the incidence of references to operators and operands, which are useful for understanding various aspects of software complexity and maintainability, namely:

- **Vocabulary**: The total unique operands and operators in a program.
- **Length**: The total value of operand and operator occurrences in a program.
- **Volume**: Represents the size of the implementation in terms of content.
- **Difficulty**: The difficulty in understanding the program.
- **Effort**: The effort required to understand and implement the program.
- **Level**: The level of abstraction of the implementation, an inverse of the difficulty.
- **Time**: The estimated time required to understand and implement the program, in seconds.
- **Bugs Delivered**: The estimation of the number of bugs.

A few other interesting metrics are:

- **Cyclomatic Complexity**: This metric allows analysing the number of independent paths that exist in a software system, allowing for the analysis of the structuredness of a program and understanding how many software tests will be required to cover all possible paths. [Fer19]
- **Cognitive Complexity**: It's a metric that analyses the mental effort required to understand and maintain a program, being especially useful in object-oriented systems to measure cohesion and class complexity. [VR18]
- **Lack of Cohesion in Methods (LCOM)**: It's the difference of null intersections with non-null intersections of methods, where higher cohesion results in a higher quality. [SA20]
- **Maintainability**: An overall measure of how easily the program can be maintained and modified, using previously mentioned metrics to calculate such as Halstead Volume and Cyclomatic Complexity, among others. [Kuk]

Having explored the background knowledge that lays the foundations of our theme and allows us to build upon it for our work, we move to detail the current advancements in the fields that concern us, namely in terms of MSR, refactoring activity detection, and refactoring recommendation tools.

2.3 Mining Software Repositories for Refactoring

Software repositories contain a large amount of information that can be used for a variety of purposes. Even though numerous methods and tools for MSR exist, we'll focus on the ones that are more suited to the context of this work, namely those that are related in some way to refactoring.

2.3.1 Tools

Dagenais and Robillard [DR11] proposed SemDiff, a recommendation system that suggested adaptations to client programs by analysing how a framework was adapted to its own changes, of which an overview is shown in fig. 2.3. In order to build its recommender, SemDiff analyses a framework's source code repository with three steps: retrieving the files and change data for each framework version; change analysis, where they performed two analyses, one that produces a list of all methods, fields, and classes that were added, removed, and modified, using an abstract syntax tree (AST) for comparison purposes, and another that finds the calls that were added or removed between two versions for each method identified previously; lastly, they store their results in a database. Whenever a developer queries the recommender with a call for potential replacements, it then gives a response based on the changes analysed previously.

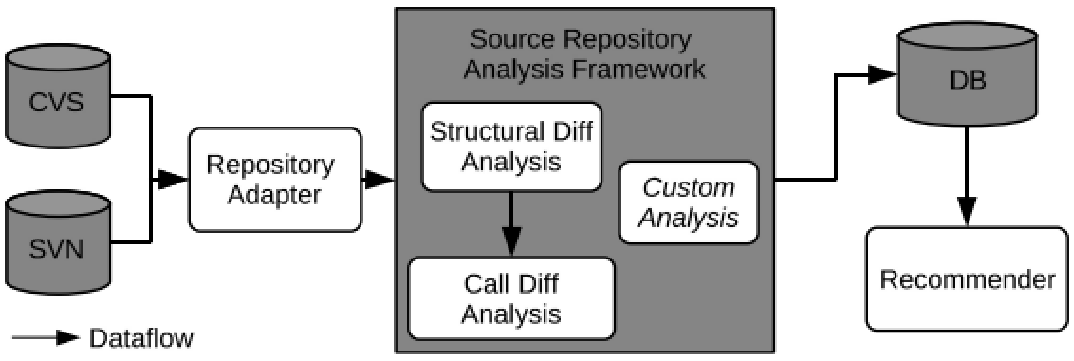


Figure 2.3: SemDiff overview (Extracted from [DR11])

Sousa et al. [SRA$^+$22] proposed SysRepoAnalysis, a tool that analyses software repositories. As seen in fig. 2.4, this tool uses a web application and message broker to analyse the repository's source code asynchronously, allowing users to execute their approach fully automatically. It has four main uses: extract commits and files, which provides information from Git repositories such as commits and modifications; calculate software metrics, such as the cyclomatic complexity and frequency of occurrence of files in commits over time; analysis of critical files, using metrics to show the list of files that have the highest frequency of commits and the highest number of lines modified over time; and generate treemaps with heatmaps based on the cyclomatic complexity of all files.

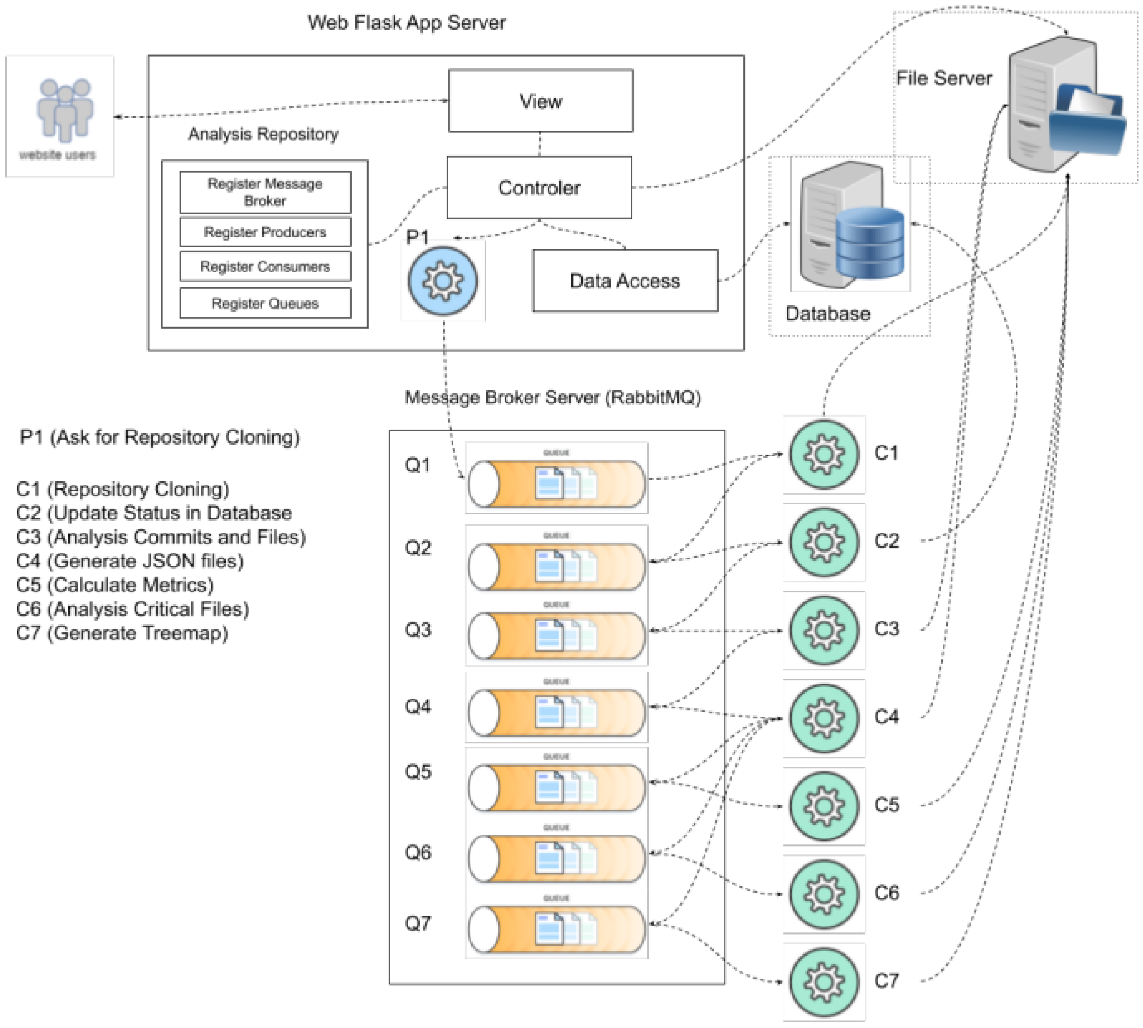


Figure 2.4: SysRepoAnalysis overview (Extracted from [SRA+22])

Sager et al. [SBPK06] proposed an approach to detect similar Java classes based on their AST representations via an MSR approach, allowing them, aside from detecting similar classes, to detect the before and after versions of a refactored class. To achieve this, they followed an approach described in fig. 2.5, which is comprised of two large steps: first, it transforms the AST representations of the classes' source code into an intermediary model, and second, it calculates the similarity between these trees using tree similarity algorithms. Their analysis can be helpful during development, especially regarding code reuse, clone detection, and faster prototyping.

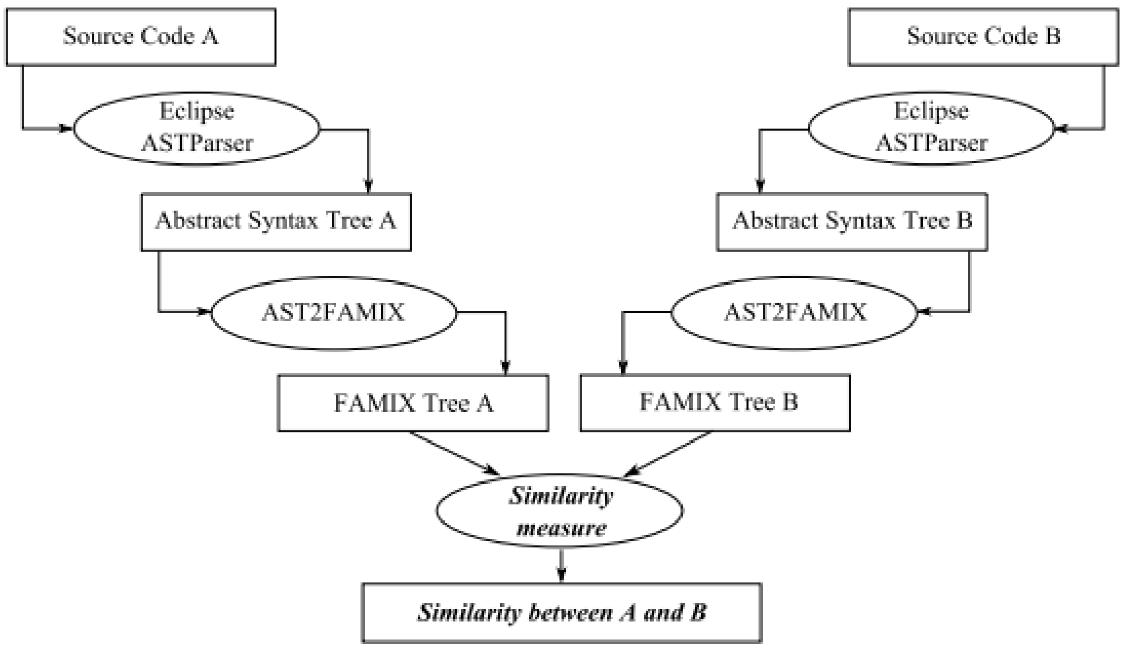


Figure 2.5: Steps to detect similar Java classes (Extracted from [SBPK06])

2.3.2 MSR Improvements

Even though many techniques and methods for MSR studies have been developed, there is always room for improvement. This section presents research ways to improve the MSR practice, specifically in the context we're looking for.

Steinhauer and Palomba [SP20] proposed DISDRILLEY, a distributed approach that can be exploited for data extraction activities related to MSR, though they exemplify it by learning software metrics to predict the effect of refactoring operations on code quality. Their approach uses a MapReduce-inspired pattern for data distribution, as it targets big data analysis and calculations by splitting the computation process into a map and a reduce phase on a cluster of computing nodes, and Apache Spark for pipeline implementation, as it increases processing speed and has good fault tolerance. They found that their approach could increase performance in MSR and outline possible improvements to optimise the approach and make it more flexible.

Wen et al. [WNLB21] performed a study on how quick remedy commits, more specifically reverted commits, which is when a developer reverts, completely or partially, the code changes in the previous commit, impact MSR practices. Specifically, towards refactoring detection practices, they found that considering reverted commits during data collecting had an impact on 97 out of the 100 systems they identified, with an average impact for a single reverted commit of 0.27 noisy data points. As such, they believe that cleaning these reverted commits is highly recommended during MSR studies regarding refactoring operations.

2.4 Refactoring Activity Detection

In order to obtain refactoring data, it will be required of us to first be able to identify where it happened. This detection has been performed in many different ways, differing in the source and methods used [OM16]. In this section, we'll present different sources and methods that make use of them.

2.4.1 Mining Commit Logs

Commit logs retain the message the developer leaves whenever they perform a commit, something that the following approaches have used as a way to detect whether a refactoring happened in that commit or not.

AlOmar et al. (2019) [AMO], in an exploratory study, found that refactorings may be self-reported by developers in their commit messages on versioned repositories, even possibly mentioning specific improvements that have been made, and obtaining a list of Self Affirmed Refactoring (SAR) patterns that could indicate refactoring activity. Later on, AlOmar et al. (2022) [APM+22] categorized how exactly developers document their refactorings via text mining in order to identify textual patterns on those commit messages.

Marmolejos et al. [MAM+22] devised an approach to automatically detect refactoring activity on Java projects that relies on text-mining, natural language preprocessing and supervised machine learning techniques. To do so, they built a two-class refactoring classifier based on SAR patterns previously obtained by AlOmar et al. (2019) [AMO] and modified them, as shown in fig. 2.6a, and new ones they themselves created, in fig. 2.6b. This classifier is then able to predict whether a commit message indicates a refactoring or not, and, after testing on different classifiers such as Bayes point machine, averaged perceptron, logistic regression, boosted decision tree, and neural network, was able to achieve an accuracy of 0.96 on the high-end.

(1) Modif*	(15) Fix* quality flaw
(2) Simplif*	(16) Remov* dependency
(3) Polish* code	(17) Code improvement*
(4) Chang* design	(18) Fix* quality issue
(5) Us* less code	(19) Renam* consistency
(6) Simplif* code	(20) Reorganiz* structure
(7) Pull* some code	(21) Fix* technical debt
(8) Us* better name	(22) Remov* unused classes
(9) Code cosmetic*	(23) Remov* redundant code
(10) Delet* old stuff	(24) Improv* code quality
(11) Simplif* design	(25) Mov* more code out of
(12) Fix* code smell	(26) Fix* naming convention
(13) Nam* improvement	(27) Chang* package structure
(14) Modulariz* class	(28) Improv* naming

(a) List of modified SAR patterns

(1) Typo	(15) Formatted
(2) Tidy*	(16) Cleaned up
(3) Spell* code	(17) Code clean
(4) Tidied	(18) Get rid of
(5) Polish*	(19) Getting rid of
(6) Clarif*	(20) Meaningful
(7) Separat*	(21) Modulariz*
(8) Optimiz*	(22) Pulled out
(9) Organiz*	(23) Cleaning up
(10) Clean-up	(24) Better name
(11) Pull out	(25) Pulling out
(12) Structur*	(26) New structure
(13) Correct*	(27) Duplicate code
(14) Normaliz*	

(b) List of new SAR patterns

Figure 2.6: Lists of SAR patterns (Extracted from [MAM+22])

Krasniqi and Cleland-Huang [KCH] presented CMMiner, a tool that detects code commit messages containing refactoring information, being able to differentiate between 12 different refactoring types. CMMiner's development followed the steps described in fig. 2.7, where the authors collected commit data, manually annotated them as refactoring or not, with the specific type of refactoring if so, followed by using the Stanford parser to tag commit messages, then extracting dependency triplets, finishing with the construction of several models. They experimented with naive Bayes, logistic regression, support vector machine, and k-nearest neighbours, achieving a precision of 0.983 and recall of 0.580 with naive Bayes in the identification of refactoring related messages.

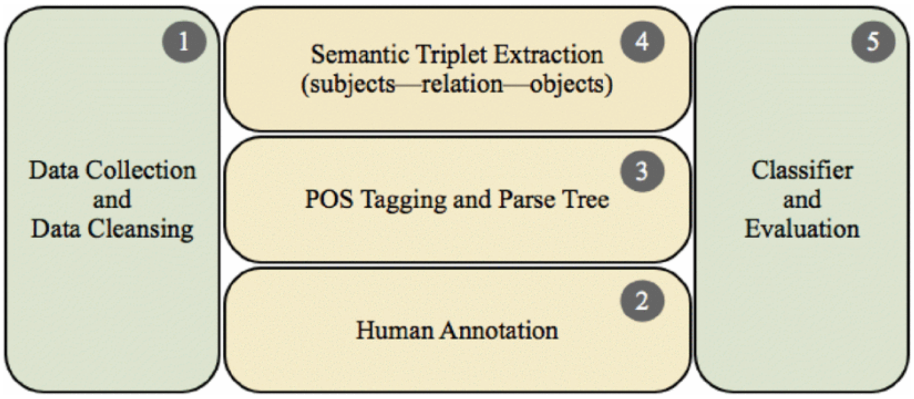

1
Data Collection and Data Cleansing
4
Semantic Triplet Extraction (subjects—relation—objects)
3
POS Tagging and Parse Tree
2
Human Annotation
5
Classifier and Evaluation

Figure 2.7: CMMiner steps (Extracted from [KCH])

2.4.2 Mining the Source Code

Aside from the commit logs, the source code itself, stored in a versioning system that stores the different versions of the code, can be an input for approaches to detecting refactoring activity.

Moser et al. [MPSS], in order to identify refactoring activity, defined a number of source code metrics, described in fig. 2.8, which, when refactorings are performed, are changed. Furthermore, they grouped several source code metrics changed into refactoring change patterns, which will have a higher probability of being induced by refactoring than other activities. When analysed between different source code versions, this can lead to the identification of refactoring activity.

Silva and Valente [SV] presented RefDiff, an automated approach that identifies refactorings performed between two versions in a git repository based on static analysis and code similarity. Their detection algorithm has two main phases: source code analysis, where two models are built from the analysis of the source code, one for the system before changes and one after changes, containing information about types, methods and fields contained in the source code; relationship analysis, which finds relationships between the two models by building a bipartite graph with two sets of vertices, code entities before and after changes, and edges connecting them that represent which refactoring occurred. RefDiff uses a formula to calculate the similarity between two code entities and, to identify which refactoring had occurred, a set of conditions is shown in fig. 2.9.

Metric name	Definition
CK metrics	Chidamber and Kemerer set of object-oriented design metrics
LOC	Number of Java statements per class
NOM	Number of methods declared in a class
NOA	Number of attributes declared in a class
LOC_PER_METHOD	Average number of Java statements per method in a class
MAX_LOC_PER_METHOD	Maximum number of Java statements per method of all methods in a class
PARAM_PER_METHOD	Average number of parameters per method in a class
MAX_PARAM_PER_METHOD	Maximum number of parameters per method of all methods in a class
MCC	Average McCabe's cyclomatic complexity per method in a class
MAX_MCC	Maximum McCabe's complexity per method of all methods in a class
NUMBER_OF_CLASSES	Number of classes

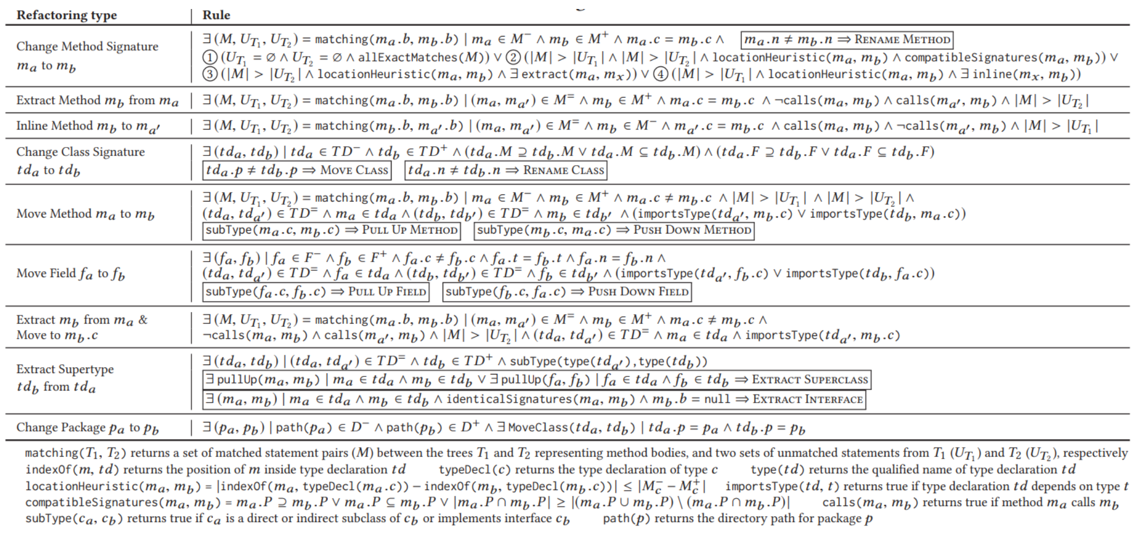
Figure 2.8: Selected metrics for computing change patterns (Extracted from [MPSS])

Relationship	Condition
	$(t_b, t_a) \in T_b \times T_a$, such that:
Same Type	$\mathrm{name}(t_b) = \mathrm{name}(t_a) \wedge \pi(t_b) \sim \pi(t_a)$
Rename Type	$\mathrm{name}(t_b) \neq \mathrm{name}(t_a) \wedge \pi(t_b) \sim \pi(t_a) \wedge \mathrm{sim}(t_b, t_a) > \tau$
Move Type	$\mathrm{name}(t_b) = \mathrm{name}(t_a) \wedge \pi(t_b) \nsim \pi(t_a) \wedge \mathrm{sim}(t_b, t_a) > \tau$
Move and Rename Type	$\mathrm{name}(t_b) \neq \mathrm{name}(t_a) \wedge \pi(t_b) \nsim \pi(t_a) \wedge \mathrm{sim}(t_b, t_a) > \tau$
Extract Supertype	$(\nexists x \in T_b \mid x \sim t_a) \wedge (\exists y \in T_a \mid t_b \sim y \wedge \mathrm{subtype}(y, t_a)) \wedge \mathrm{sim_p}(t_a, t_b) > \tau$
	$(m_b, m_a) \in M_b \times M_a$, such that:
Same Method	$\mathrm{sig}(m_b) = \mathrm{sig}(m_a) \wedge \pi(m_b) \sim \pi(m_a)$
Rename Method	$\mathrm{name}(m_b) \neq \mathrm{name}(m_a) \wedge \pi(m_b) \sim \pi(m_a) \wedge \mathrm{sim}(m_b, m_a) > \tau$
Change Method Signature	$\mathrm{name}(m_b) = \mathrm{name}(m_a) \wedge \mathrm{sig}(m_b) \neq \mathrm{sig}(m_a) \wedge \pi(m_b) \sim \pi(m_a) \wedge \mathrm{sim}(m_b, m_a) > \tau$
Pull Up Method	$\mathrm{sig}(m_b) = \mathrm{sig}(m_a) \wedge \mathrm{subtype}(\pi(m_b)^{\sim}, \pi(m_a)) \wedge \mathrm{sim}(m_b, m_a) > \tau$
Push Down Method	$\mathrm{sig}(m_b) = \mathrm{sig}(m_a) \wedge \mathrm{supertype}(\pi(m_b)^{\sim}, \pi(m_a)) \wedge \mathrm{sim}(m_b, m_a) > \tau$
Move Method	$\mathrm{name}(m_b) = \mathrm{name}(m_a) \wedge \pi(m_b) \nsim \pi(m_a) \wedge \neg \mathrm{subOrSuper}(\pi(m_b)^{\sim}, \pi(m_a)) \wedge \mathrm{sim}(m_b, m_a) > \tau$
Extract Method	$(\nexists x \in M_b \mid x \sim m_a) \wedge (\exists y \in M_a \mid m_b \sim y \wedge y \in \mathrm{callers}(m_a)) \wedge \mathrm{sim_p}(m_a, m_b) > \tau$
Inline Method	$(\nexists x \in M_a \mid m_b \sim x) \wedge (\exists y \in M_b \mid y \sim m_a \wedge y \in \mathrm{callers}(m_b)) \wedge \mathrm{sim_p}(m_b, m_a) > \tau$
	$(f_b, f_a) \in F_b \times F_a$, such that:
Same Field	$\mathrm{name}(f_b) = \mathrm{name}(f_a) \wedge \mathrm{type}(f_b) = \mathrm{type}(f_a) \wedge \pi(f_b) \sim \pi(f_a)$
Pull Up Field	$\mathrm{name}(f_b) = \mathrm{name}(f_a) \wedge \mathrm{type}(f_b) = \mathrm{type}(f_a) \wedge \mathrm{subtype}(\pi(f_b)^{\sim}, \pi(f_a)) \wedge \mathrm{sim}(f_b, f_a) > \tau$
Push Down Field	$\mathrm{name}(f_b) = \mathrm{name}(f_a) \wedge \mathrm{type}(f_b) = \mathrm{type}(f_a) \wedge \mathrm{supertype}(\pi(f_b)^{\sim}, \pi(f_a)) \wedge \mathrm{sim}(f_b, f_a) > \tau$
Move Field	$\mathrm{name}(f_b) = \mathrm{name}(f_a) \wedge \mathrm{type}(f_b) = \mathrm{type}(f_a) \wedge \pi(f_b) \nsim \pi(f_a) \wedge \neg \mathrm{subOrSuper}(\pi(f_b)^{\sim}, \pi(f_a)) \wedge \mathrm{sim}(f_b, f_a) > \tau$

$\mathrm{name}(e)$	simple name of a code entity e	$\pi(e)$	container entity of a code entity e (it may be a type or a package)
$\mathrm{sig}(m)$	signature of a method m	$e_1 \sim e_2$	exists a matching relationship between e_1 and e_2
$\mathrm{type}(f)$	type of a field f	$e_1 \nsim e_2$	does not exists a matching relationship between e_1 and e_2
$\mathrm{subtype}(t_1, t_2)$	t_1 is subtype of t_2	$e^{\sim}$	the code entity that matches with e after the change
$\mathrm{supertype}(t_1, t_2)$	t_1 is supertype of t_2	$\mathrm{callers}(m_a)$	the set of methods that call m_a
$\mathrm{subOrSuper}(t_1, t_2)$	$\mathrm{subtype}(t_1, t_2) \vee \mathrm{supertype}(t_1, t_2)$	$\mathrm{sim}(e_1, e_2)$	similarity index between e_1 and e_2
		$\mathrm{sim_p}(e_1, e_2)$	similarity index between e_1 and e_2 for non-matching relationships

Figure 2.9: Relationship Types for RefDiff (Extracted from [SV])

Tsantalis et al. [TME^{+}18] proposed RMINER, an AST-based statement matching algorithm which determines refactoring candidates without requiring any code similarity thresholds to operate. Their tool takes two revisions as input and starts by representing all the information it extracts from the two revisions, including the type declarations, the set of fields and methods inside the type declarations, and the set of all directories, using the Eclipse Java Development Tools AST Parser. RMINER then apply their statement matching algorithm, a bottom-up approach. Lastly, it performs the refactoring detection, as it supports 15 refactoring types, in two phases: first, it tries and match based on the code elements' signatures; secondly, if it couldn't match them before, it examines the rest of the code elements by applying, in order, the set of rules shown in fig. 2.10. According to the authors' analysis, this tool achieves 98% precision, and 87% recall, while having a very small computation cost of 58 ms per commit on average.



Refactoring type	Rule
Change Method Signature m_a to m_b	$\exists(M, U_{T_1}, U_{T_2}) = \texttt{matching}(m_a.b, m_b.b) \mid m_a \in M^- \wedge m_b \in M^+ \wedge m_a.c = m_b.c \wedge$ $\boxed{m_a.n \neq m_b.n \Rightarrow \text{RENAME METHOD}}$ ① $(U_{T_1} = \varnothing \wedge U_{T_2} = \varnothing \wedge \texttt{allExactMatches}(M)) \vee$ ② $(\lvert M\rvert > \lvert U_{T_1}\rvert \wedge \lvert M\rvert > \lvert U_{T_2}\rvert \wedge \texttt{locationHeuristic}(m_a, m_b) \wedge \texttt{compatibleSignatures}(m_a, m_b)) \vee$ ③ $(\lvert M\rvert > \lvert U_{T_2}\rvert \wedge \texttt{locationHeuristic}(m_a, m_b) \wedge \exists\, \texttt{extract}(m_a, m_x)) \vee$ ④ $(\lvert M\rvert > \lvert U_{T_1}\rvert \wedge \texttt{locationHeuristic}(m_a, m_b) \wedge \exists\, \texttt{inline}(m_x, m_b))$
Extract Method m_b from m_a	$\exists(M, U_{T_1}, U_{T_2}) = \texttt{matching}(m_a.b, m_b.b) \mid (m_a, m_{a'}) \in M^= \wedge m_b \in M^+ \wedge m_a.c = m_b.c \wedge \neg\texttt{calls}(m_a, m_b) \wedge \texttt{calls}(m_{a'}, m_b) \wedge \lvert M\rvert > \lvert U_{T_2}\rvert$
Inline Method m_b to $m_{a'}$	$\exists(M, U_{T_1}, U_{T_2}) = \texttt{matching}(m_b.b, m_{a'}.b) \mid (m_a, m_{a'}) \in M^= \wedge m_b \in M^- \wedge m_{a'}.c = m_b.c \wedge \texttt{calls}(m_a, m_b) \wedge \neg\texttt{calls}(m_{a'}, m_b) \wedge \lvert M\rvert > \lvert U_{T_1}\rvert$
Change Class Signature td_a to td_b	$\exists(td_a, td_b) \mid td_a \in TD^- \wedge td_b \in TD^+ \wedge (td_a.M \supseteq td_b.M \vee td_a.M \subseteq td_b.M) \wedge (td_a.F \supseteq td_b.F \vee td_a.F \subseteq td_b.F)$ $\boxed{td_a.p \neq td_b.p \Rightarrow \text{MOVE CLASS}}$ $\boxed{td_a.n \neq td_b.n \Rightarrow \text{RENAME CLASS}}$
Move Method m_a to m_b	$\exists(M, U_{T_1}, U_{T_2}) = \texttt{matching}(m_a.b, m_b.b) \mid m_a \in M^- \wedge m_b \in M^+ \wedge m_a.c \neq m_b.c \wedge \lvert M\rvert > \lvert U_{T_1}\rvert \wedge \lvert M\rvert > \lvert U_{T_2}\rvert \wedge$ $(td_a, td_{a'}) \in TD^= \wedge m_a \in td_a \wedge (td_b, td_{b'}) \in TD^= \wedge m_b \in td_{b'} \wedge (\texttt{importsType}(td_{a'}, m_b.c) \vee \texttt{importsType}(td_b, m_a.c))$ $\boxed{\texttt{subType}(m_a.c, m_b.c) \Rightarrow \text{PULL UP METHOD}}$ $\boxed{\texttt{subType}(m_b.c, m_a.c) \Rightarrow \text{PUSH DOWN METHOD}}$
Move Field f_a to f_b	$\exists(f_a, f_b) \mid f_a \in F^- \wedge f_b \in F^+ \wedge f_a.c \neq f_b.c \wedge f_a.t = f_b.t \wedge f_a.n = f_b.n \wedge$ $(td_a, td_{a'}) \in TD^= \wedge f_a \in td_a \wedge (td_b, td_{b'}) \in TD^= \wedge f_b \in td_{b'} \wedge (\texttt{importsType}(td_{a'}, f_b.c) \vee \texttt{importsType}(td_b, f_a.c))$ $\boxed{\texttt{subType}(f_a.c, f_b.c) \Rightarrow \text{PULL UP FIELD}}$ $\boxed{\texttt{subType}(f_b.c, f_a.c) \Rightarrow \text{PUSH DOWN FIELD}}$
Extract m_b from m_a & Move to $m_b.c$	$\exists(M, U_{T_1}, U_{T_2}) = \texttt{matching}(m_a.b, m_b.b) \mid (m_a, m_{a'}) \in M^= \wedge m_b \in M^+ \wedge m_a.c \neq m_b.c \wedge$ $\neg\texttt{calls}(m_a, m_b) \wedge \texttt{calls}(m_{a'}, m_b) \wedge \lvert M\rvert > \lvert U_{T_2}\rvert \wedge (td_a, td_{a'}) \in TD^= \wedge m_a \in td_a \wedge \texttt{importsType}(td_{a'}, m_b.c)$
Extract Supertype td_b from td_a	$\exists(td_a, td_b) \mid (td_a, td_{a'}) \in TD^= \wedge td_b \in TD^+ \wedge \texttt{subType}(\texttt{type}(td_{a'}), \texttt{type}(td_b))$ $\boxed{\exists\, \texttt{pullUp}(m_a, m_b) \mid m_a \in td_a \wedge m_b \in td_b \vee \exists\, \texttt{pullUp}(f_a, f_b) \mid f_a \in td_a \wedge f_b \in td_b \Rightarrow \text{EXTRACT SUPERCLASS}}$ $\boxed{\exists(m_a, m_b) \mid m_a \in td_a \wedge m_b \in td_b \wedge \texttt{identicalSignatures}(m_a, m_b) \wedge m_b.b = \texttt{null} \Rightarrow \text{EXTRACT INTERFACE}}$
Change Package p_a to p_b	$\exists(p_a, p_b) \mid \texttt{path}(p_a) \in D^- \wedge \texttt{path}(p_b) \in D^+ \wedge \exists\, \texttt{MoveClass}(td_a, td_b) \mid td_a.p = p_a \wedge td_b.p = p_b$

$\texttt{matching}(T_1, T_2)$ returns a set of matched statement pairs (M) between the trees T_1 and T_2 representing method bodies, and two sets of unmatched statements from T_1 (U_{T_1}) and T_2 (U_{T_2}), respectively
$\texttt{indexOf}(m, td)$ returns the position of m inside type declaration td $\texttt{typeDecl}(c)$ returns the type declaration of type c $\texttt{type}(td)$ returns the qualified name of type declaration td
$\texttt{locationHeuristic}(m_a, m_b) = |\texttt{indexOf}(m_a, \texttt{typeDecl}(m_a.c)) - \texttt{indexOf}(m_b, \texttt{typeDecl}(m_b.c))| \leq |M_c^- - M_c^+|$ $\texttt{importsType}(td, t)$ returns true if type declaration td depends on type t
$\texttt{compatibleSignatures}(m_a, m_b) = m_a.P \supseteq m_b.P \vee m_a.P \subseteq m_b.P \vee |m_a.P \cap m_b.P| \geq |(m_a.P \cup m_b.P) \setminus (m_a.P \cap m_b.P)|$ $\texttt{calls}(m_a, m_b)$ returns true if method m_a calls m_b
$\texttt{subType}(c_a, c_b)$ returns true if c_a is a direct or indirect subclass of c_b or implements interface c_b $\texttt{path}(p)$ returns the directory path for package p

Figure 2.10: Refactoring detection rules in RMiner (Extracted From [TME+18])

Detecting refactoring activity is paramount to allows us to build our intended knowledge base, though the there still is a need to explore tools that could make use of such knowledge, or those who provide recommendations based on alternate methods.

2.5 Refactoring Recommendation Tools

As refactoring is a process that proves itself crucial to the maintainability of software systems, several tools have been created to identify and recommend refactoring opportunities to developers using different sources for its purpose, some of which we will identify and describe in this section.

2.5.1 Analysing the Source Code

Just like it was shown before, the source code was a source to identify previous refactorings, and it can also be used by approaches that inspect it through various means and identify code fragments that should be refactored.

Foster et al. [FGL12] proposed WitchDoctor, a tool which can detect, live, when a programmer is performing a refactoring operation and then propose the completed refactoring to them long before they finish it. As shown in fig. 2.11, WitchDoctor is composed of three components: change detection, using a technique that initially bypasses AST to achieve interactive speed and to be able to tolerate non-parseable program states; specification matching, using the Rete algorithm, an efficient pattern-matching algorithm in production-rule systems; and refactoring execution, presenting the refactoring prediction to the user and writing it, if the user wants. This system works for certain refactoring operations: Rename Variable, Rename Class, Rename Method, Extract Local Variable, and Extract Method.

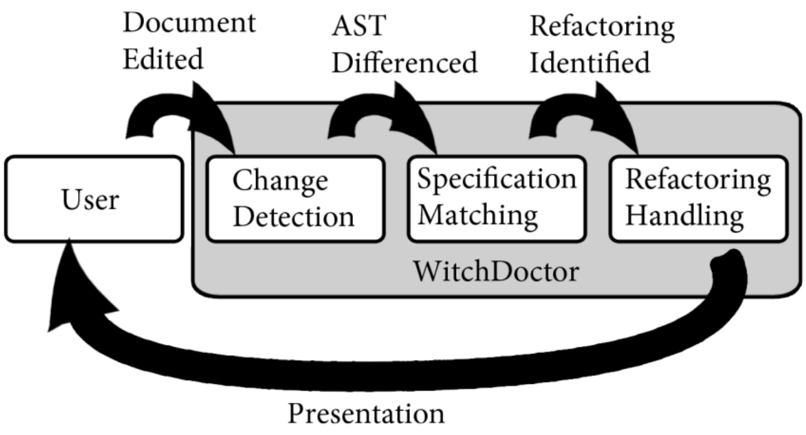


Figure 2.11: Graphical representation of WitchDoctor's workflow (Extracted From [FGL12])

Yoshida et al. [YNCI] proposed a system to provide code clone recommendations after a user executes an Extract Method refactoring. As can be seen in fig. 2.12, they provide live suggestions after an Extract Method refactoring is performed by analysing the AST and finding possible code clones that can benefit from the same refactoring.

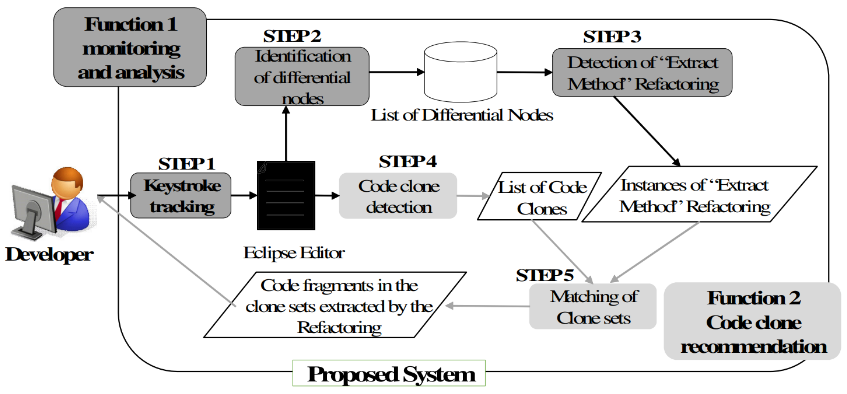

Function 1 monitoring and analysis
STEP2 Identification of differential nodes
List of Differential Nodes
STEP3 Detection of 'Extract Method' Refactoring
STEP1 Keystroke tracking
Eclipse Editor
STEP4 Code clone detection
List of Code Clones
Instances of 'Extract Method' Refactoring
Developer
Code fragments in the clone sets extracted by the Refactoring
STEP5 Matching of Clone sets
Function 2 Code clone recommendation
Proposed System

Figure 2.12: Overview of the proposed system (Extracted From [YNCI])

JS CodeFormer [Ste], created by Stead, is a refactoring and code automation Visual Studio

Code extension for JavaScript, which supports refactorings such as Extract Method, Extract Variable to Parameter, Extract Variable, Inline Variable, Invert If Statement, Invert Ternary Expression, and Rename, checking the code for possible refactorings and performing them automatically, should the user want it to.

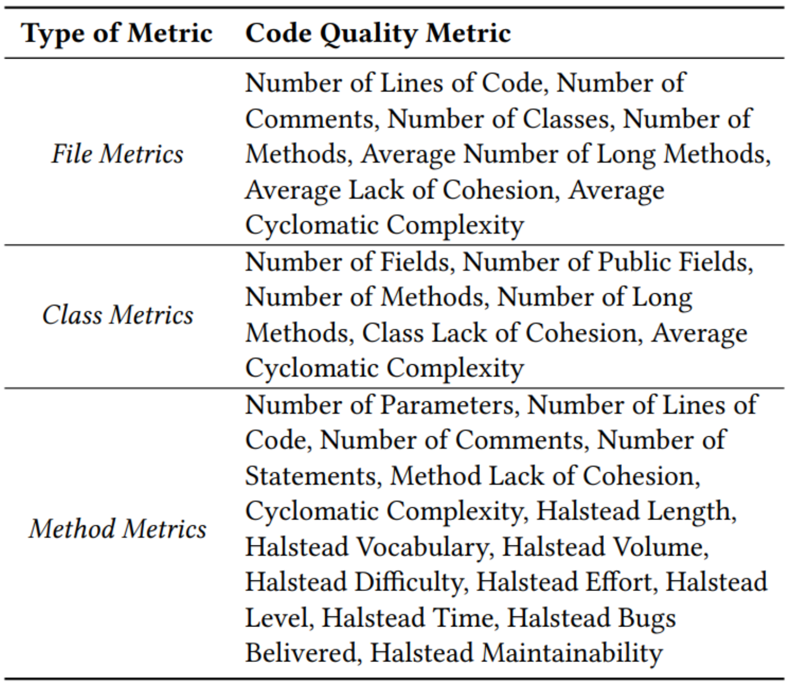
Franklin et al. [FGLD] proposed LAMBDAFICATOR, a Java tool that automates two refactorings: AnonymousToLambda, which converts anonymous inner classes to lambda expressions, and ForLoopToFunctional, which converts for loops into functional operators that used lambda expressions. It is a live programming approach that provides refactoring recommendations as the programmer codes, while also allowing for batch refactorings on existent code.

Dipongkor et al. [DAN] proposed an approach that considers the frequency of coupled elements, whether they be methods and/or attributes, to provide Move Method recommendations.

glean [Wix] is a Visual Studio Code extension that reviews the source code and performs refactorings for React, JavaScript, and TypeScript. It suggests refactorings such as extracting code JSX into new React components, conversion of class components to functional components, and vice-versa, renaming state variables, and extracting code to files.

2.5.2 Code Quality Metrics

Several tools also make use of code quality metrics, where they can identify specific metrics and, by their values, possibly identify refactoring opportunities.



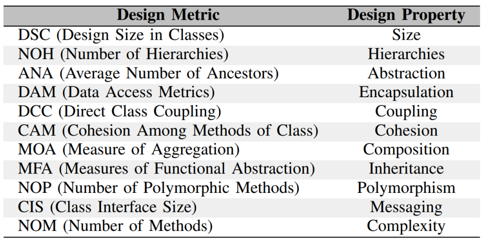
Type of Metric	**Code Quality Metric**
File Metrics	Number of Lines of Code, Number of Comments, Number of Classes, Number of Methods, Average Number of Long Methods, Average Lack of Cohesion, Average Cyclomatic Complexity
Class Metrics	Number of Fields, Number of Public Fields, Number of Methods, Number of Long Methods, Class Lack of Cohesion, Average Cyclomatic Complexity
Method Metrics	Number of Parameters, Number of Lines of Code, Number of Comments, Number of Statements, Method Lack of Cohesion, Cyclomatic Complexity, Halstead Length, Halstead Vocabulary, Halstead Volume, Halstead Difficulty, Halstead Effort, Halstead Level, Halstead Time, Halstead Bugs Belivered, Halstead Maintainability

Figure 2.13: Summary of code quality metrics supported by LiveRef (Extracted from [FAR22])

Fernandes et al. [FAR22] [FAR23] proposed LiveRef, a plugin for IntelliJ to support live refactoring in Java systems, focused on the Extract Method refactoring, which can provide real-time refactoring recommendations. Their approach is based on a number of code quality metrics, shown in fig. 2.13. They start by identifying code fragments that could be extracted and joining consecutive ones into code blocks that could potentially result in extracted methods, based on defined conditions such as concurrent statements, number of statements, and percentage of statements of the original method. After this selection, they sort their candidates by severity, trying to minimize the cyclomatic complexity first and, after, reduce the number of statements and lack of cohesion. The suggestions are presented visually to the user, who can, by clicking, perform the refactorings automatically.

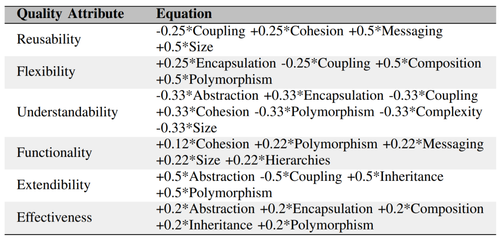
Bavota et al. [BDLM^{+}] developed ARIES, an Eclipse plugin, focused on the Extract Class refactoring in order to address the Blob antipattern. The tool starts by identifying candidate Blobs, through the use of a threshold on the values of three metrics: LCOM, Class Cohesion Coupling, and Message Passing Coupling, using cohesion and coupling metrics to evaluate the quality of a class. Having a candidate class selected by a user, they can get further insight into it and the tool suggests the methods to be extracted for the Extract Class refactoring, which can be automatically done with the user's input.

Couto et al. [SCRT17] a semi-automatic software restructuring approach based on quality attributes. This approach provides Move Method recommendations based on the Quality Model for Object Oriented Design (QMOOD), a model that uses 11 object-oriented design properties, shown in fig. 2.14a and calculates 6 quality attributes based on them, through the equations described in fig. 2.14b.



Design Metric	Design Property
DSC (Design Size in Classes)	Size
NOH (Number of Hierarchies)	Hierarchies
ANA (Average Number of Ancestors)	Abstraction
DAM (Data Access Metrics)	Encapsulation
DCC (Direct Class Coupling)	Coupling
CAM (Cohesion Among Methods of Class)	Cohesion
MOA (Measure of Aggregation)	Composition
MFA (Measures of Functional Abstraction)	Inheritance
NOP (Number of Polymorphic Methods)	Polymorphism
CIS (Class Interface Size)	Messaging
NOM (Number of Methods)	Complexity

(a) Design Metrics for Design Properties



Quality Attribute	Equation
Reusability	-0.25*Coupling +0.25*Cohesion +0.5*Messaging +0.5*Size
Flexibility	+0.25*Encapsulation -0.25*Coupling +0.5*Composition +0.5*Polymorphism
Understandability	-0.33*Abstraction +0.33*Encapsulation -0.33*Coupling +0.33*Cohesion -0.33*Polymorphism -0.33*Complexity -0.33*Size
Functionality	+0.12*Cohesion +0.22*Polymorphism +0.22*Messaging +0.22*Size +0.22*Hierarchies
Extendibility	+0.5*Abstraction -0.5*Coupling +0.5*Inheritance +0.5*Polymorphism
Effectiveness	+0.2*Abstraction +0.2*Encapsulation +0.2*Composition +0.2*Inheritance +0.2*Polymorphism

(b) Equations for Quality Attributes

Figure 2.14: Design Metrics and Equations for QMOOD (Extracted from [SCRT17])

Their recommender takes candidates of Extract Method refactorings, in the pair of a method and the code to extract, and assigns a probability of it being good refactoring operation before presenting it to the user

Alzahrani [Alz22] proposed an approach that considers the combination of the cohesion and the coupling of classes to identify Extract Class opportunities. This approach uses a metric that measures the cohesion of a specific class based on similarities between pairs of methods in terms of clients' usage to find methods to extract that have low cohesion, while avoiding the extract of classes with tight coupling.

Napoli et al. [NPT] proposed an approach to suggest Move Method refactoring opportunities based on modularity metrics. The approach, shown in fig. 2.15, starts by analysing the system before calculating metrics such as LCOM and Coupling Between Objects. Afterwards, the developer could select which characteristics of the system they prefer to improve or receive refactoring suggestions to improve everything.

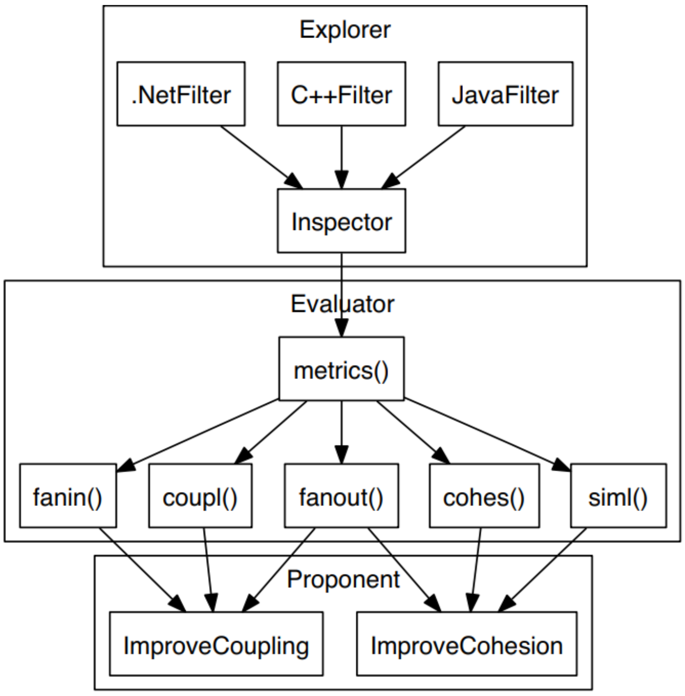


Figure 2.15: Main components of the tool for the proposed approach (Extracted from [NPT])

Charalampidou et al. [CAC$^+$17] introduced a tool that suggests Extract Method refactorings. They assess the cohesion between code fragments, identifying code that, when extracted into a new method, would improve the cohesion of the overall code.

2.6 Summary

In this chapter, we described some approaches and tools related to the topics of our dissertation. With this analysis, we intended to understand the existing techniques and their limitations.

In terms of MSR, we found approaches that analyse software repositories and two possible avenues for improvement, one for general data extraction approaches in the form of an efficient pipeline and one towards refactoring detection practices, with the discovery that reverted commits have a high impact on the analysed data.

On refactoring activity detection, we analysed two main approaches: mining the commit logs and mining the source code. For the former, the detection was mainly done through the use of classifiers, which achieved high results regardless of the method. In terms of the latter, we presented several tools that used metrics and/or relationships between different code versions to detect which refactoring had occurred, all with the same limitation of only working within a specific set of well-known refactorings.

Next, we presented approaches that recommend refactorings to a user, focusing on those that analyse the source code and that define code quality metrics. Both of these approaches come with limitations in terms of the situations where they work, namely the programming language, the refactorings they support, the intuitiveness of their use, and their adaptability to diverse software contexts.

Thus, with the analysis performed into our main topics, we believe we were able to identify and verify the main contributions towards them and the limitations that plague them, some of which we intend to address in our work.

Chapter 3

Problem Statement

3.1 Context

We are working on top of an existing refactoring recommendation system, an IntelliJ IDEA plugin called LiveRef [FAR22] [FAR23], which can identify, suggest, and apply certain refactoring operations on Java code and was developed by the second supervisor of this thesis.

This plugin makes use of a variety of code quality metrics via defined thresholds in order to make its recommendations, which comes with the limitation of not being able to adapt to ever-changing software contexts.

We will be applying our ideas to the improvement of this plugin with the idea that they can be replicated later on for similar contexts.

3.2 Research Questions

Our main purpose is to understand whether or not an approach based on real-life data incorporated into a classification model will provide better refactoring suggestions as opposed to traditional threshold-based methods.

This is to be able to provide recommendations for developers and, thus, allow them to save even more time and effort when considering refactorings for their code corpus.

As such, we propose the following hypothesis:

"A refactoring recommendation system based on a dynamic classification model, built with real-life data, will lead to more accurate suggestions when compared to threshold-based methods."

Based on this hypothesis, we intend to answer the following research questions:

RQ1 "How do threshold-based refactoring recommendations compare to the refactoring practices developers employ in real-life contexts?"

RQ2 "Is a classification model based on real data able to improve on the refactoring recommendations when compared to a threshold method?"

3.3 Proposed Approach

Considering the previously described hypothesis and research questions, we will have two major focuses:

- In order to be able to ascertain the validity of our first claim, we will be making use of the plugin before we build upon it, thus still using threshold-based methods, and collecting real-life data, comparing them to each other.
- To test our second claim, we will be using a larger amount of real-life data and using it to build a classification model which will be integrated into the plugin, replacing the original method, which should be able to give more context-accurate suggestions. Additionally, the plugin should also be customizable to the user, allowing each user to possibly be provided with different results based on their own usage patterns.

3.4 Validation

To verify whether or not our knowledge base yielded better results than the existing one, we'll be conducting an automated analysis, comparing the performance given by our knowledge base with both the base plugin and the real-life data.

Chapter 4

LiveRef Knowledge Base

Having stated what we want to achieve, we will now describe the method we used to try to reach our goals.

4.1 Context

As mentioned before, we will be building upon an existing plugin, LiveRef [FAR22] [FAR23], one that is focused on the Extract Method refactoring.

In addition to what is currently published, this tool has been further developed, and we will be using a more advanced version of it which has additional support for refactoring types, aside from Extract Method, including Extract Class, Extract Variable, Replace Inheritance with Delegation, Introduce Parameter Object, Move Method and String Comparison Replacement.

However, we will be focused especially on the Extract Method refactoring as it is both the focus of the original plugin and the one for which we believed our proposal would make the most sense, even though we tested our hypothesis on the Extract Class refactoring, too. Aside from that, the Extract Method refactoring, of the supported ones, ended up being the one for which we were able to find the most data, allowing us a more robust solution.

To illustrate better what we intend to achieve with the plugin itself, the diagrams in fig. 4.1 compare how the plugin worked originally and how we adapted it to be able to test our approach to the same problem. As previously mentioned, we will be focused on 2 refactorings, thus the diagrams only reflect those. As can be seen, the plugin's overall activity wasn't altered, only the specifics regarding how a method or class will be selected for further consideration.

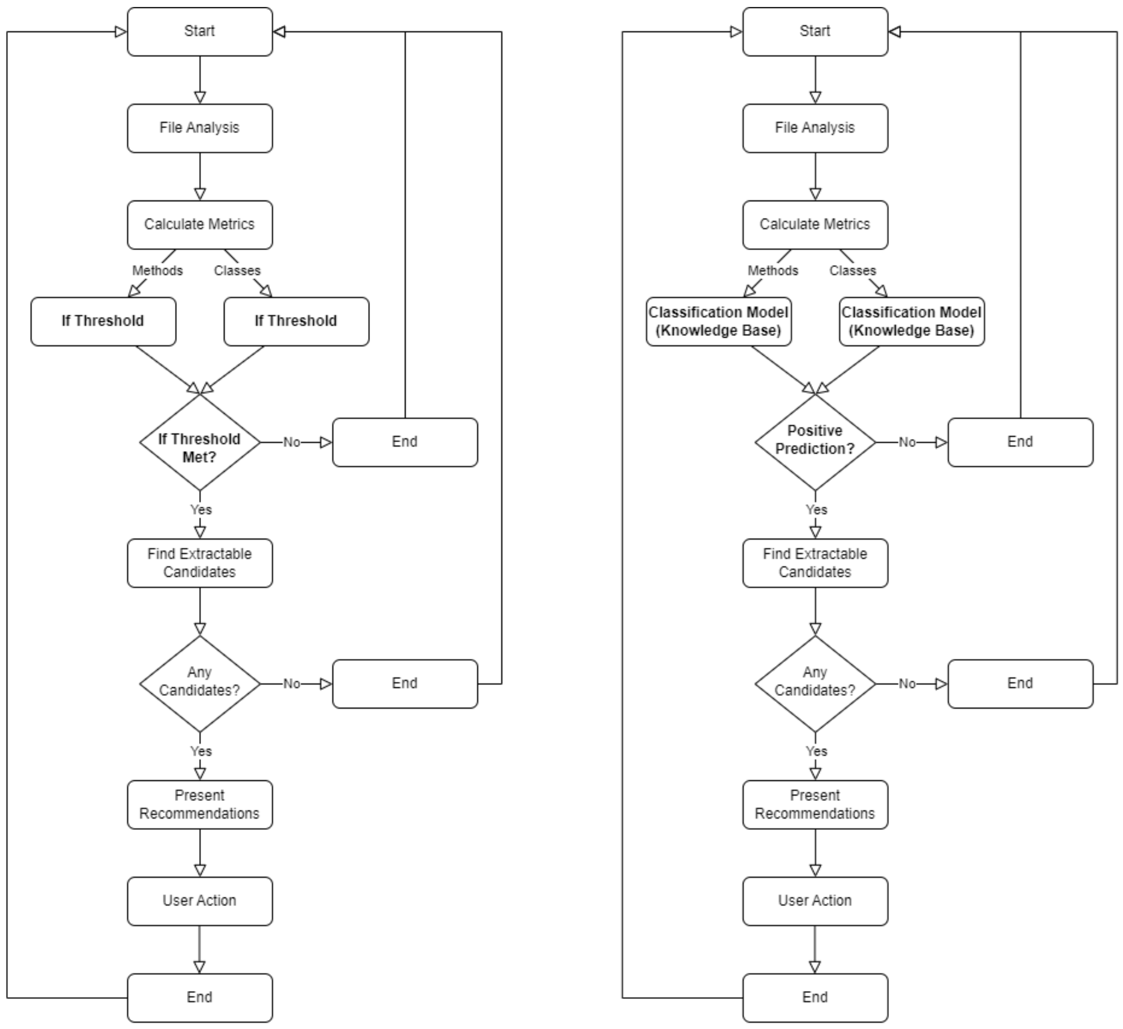


Figure 4.1: Comparison of original LiveRef (Left) with our proposed alterations (Right)

4.2 Development Strategy

To develop what we intended to, we needed to delineate a plane that would allow us to reach the objectives we needed to and be able to answer the research questions we proposed before.

This plan, further detailed in the activity diagram shown in fig. 4.2, is comprised of three major stages: preparation, classification models, and plugin.

At first, we started by collecting the data, in the form of code quality metrics, and building a database. This process required us to delve into the available metrics and select the relevant ones, for each refactoring, and build a different table for each in the database, where each row represented the metrics before and after a refactoring happened.

We then ran the original LiveRef plugin on a delineated portion of data, one that wasn't used later on for the classification model, and compared the metrics with the existing metrics from real-life data.

Afterwards, we shifted our focus towards the development of the classification models, one for each supported refactoring, which are able to select the methods or classes that should be considered for being the target of Extract Method or Extract Class refactorings, respectively.

Lastly, these models were integrated into the Plugin itself, so we can test them, and make sure they can be used and continuously update as time goes on. Aside from that, we built a an action that allows a user to provide their old software repositories to extract data to feed the model, so it can adapt quicker to the user itself.

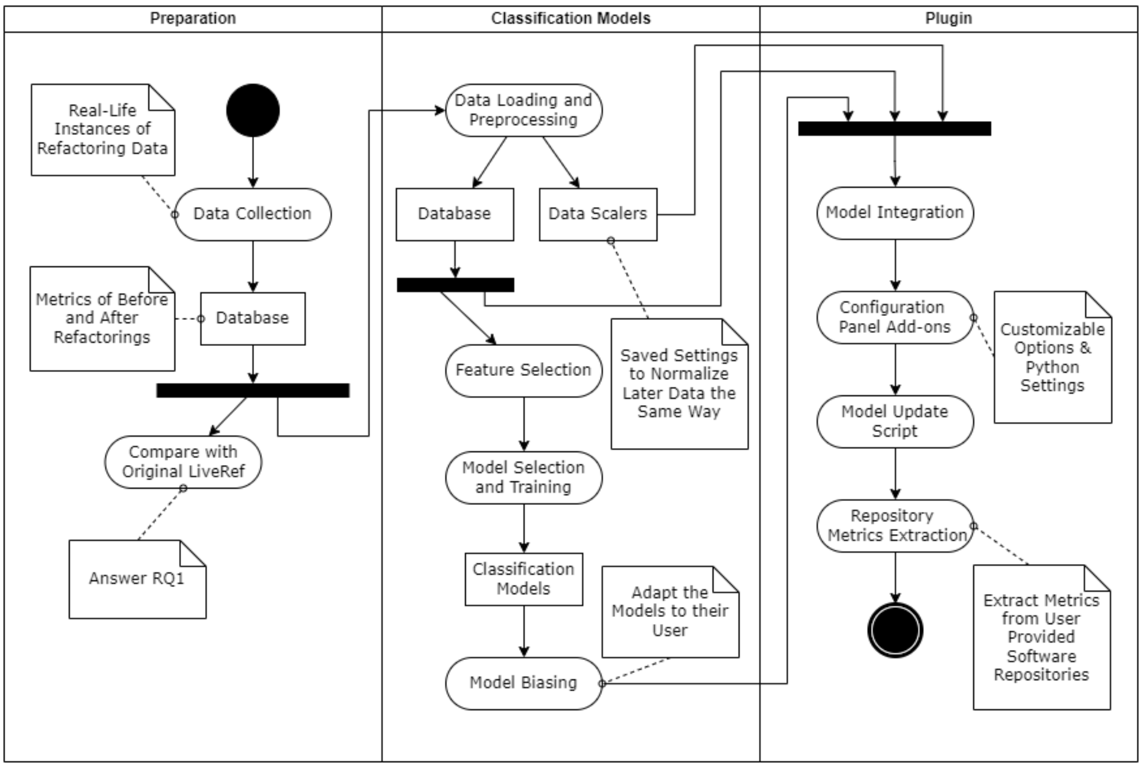

Preparation
Classification Models
Plugin
Real-Life Instances of Refactoring Data
Data Collection
Metrics of Before and After Refactorings
Database
Compare with Original LiveRef
Answer RQ1
Data Loading and Preprocessing
Database
Data Scalers
Saved Settings to Normalize Later Data the Same Way
Feature Selection
Model Selection and Training
Classification Models
Adapt the Models to their User
Model Biasing
Model Integration
Configuration Panel Add-ons
Customizable Options & Python Settings
Model Update Script
Repository Metrics Extraction
Extract Metrics from User Provided Software Repositories

Figure 4.2: Activity diagram for solution development

4.3 Data Collection

As previously mentioned, we wanted to use real-life data, specifically from software repositories, where real instances of refactoring operations were present. To do so, we explored different possibilities, which included mining different repositories and identifying and collecting the refactoring operations or using an already existing data collection.

We decided to use an already created and analysed dataset, SmartSHARK MongoDB Release 2.2 [TTH21], comprised of Java data from nearly 100 different projects, analysed and placed into a MongoDB database. Relevant to our goals are the “refactoring”, “commit”, “file_action”, and “vcs_system” collections, which, together, contain information about the commits that had refactorings occurring in them, including the type of refactoring and a description, and in the file and project where it happened.

With this data, we were able to build a Java script that would query the database to obtain the relevant information and extract, from locally cloned repositories, the relevant metrics, saving them to a SQLite database.

Thus, we were able to create a database comprised of 25000 different instances of Extract Method refactorings and 2500 instances of Extract Class refactorings, for which we collected several code quality metrics from before and after the refactoring was performed, which can be seen in table 4.1.

Metric	**Extract Method**	**Extract Class**
Number of Lines of Code	✓	✓
Number of Comments	✓	
Number of Blank Lines	✓	
Total Lines	✓	
Number of Parameters	✓	
Number of Statements	✓	
Halstead Length	✓	✓
Halstead Vocabulary	✓	✓
Halstead Volume	✓	✓
Halstead Difficulty	✓	✓
Halstead Effort	✓	✓
Halstead Level	✓	✓
Halstead Time	✓	✓
Halstead Bugs Delivered	✓	✓
Halstead Maintainability	✓	✓
Cyclomatic Complexity	✓	✓
Cognitive Complexity	✓	✓
LCOM	✓	
Number of Properties		✓
Number of Public Attributes		✓
Number of Public Methods		✓
Number of Protected Fields		✓
Number of Protected Methods		✓
Number of Long Methods		✓
Number of Methods		✓
Number of Constructors		✓

Table 4.1: Collected Metrics for Extract Method and Extract Class Refactorings

4.4 Threshold model baseline

In order to assess whether or not a threshold-based method was good at providing refactoring suggestions, we compared it to the previously mentioned data by conducting an automated analysis that would predict and enforce a refactoring suggestion by the baseline plugin on the same situations where real developers had performed it.

We extracted further 800 rows of data, to keep separate from the existing database we had created, and ran the plugin on that data. From our results, we found that the plugin was able to find opportunities for the same methods where real developers performed Extract Method refactorings 56% of the time.

We also compared the effectiveness of each refactoring by measuring code quality metrics on the code before a refactoring was applied and after, both in the real-life data and on the code originated after the plugin was used, as it can be seen in table 4.2.

Metric	**Real-Life**	**LiveRef**
Total Lines	-13.38	-9.75
Halstead Length	-11.54	-9.98
Halstead Vocabulary	-87.54	-77.02
Halstead Volume	-85.74	-75.02
Halstead Difficulty	-2.86	-2.60
Halstead Effort	-2763.38	-2440.33
Halstead Level	0.06	0.03
Halstead Time	-153.52	-135.57
Halstead Maintainability	4.94	4.03
Cyclomatic Complexity	-2.13	-1.90
Cognitive Complexity	-42.96	-13.69
LCOM	-0.04	0.001

Table 4.2: Comparison of Average Differences for Various Metrics when Compared to Baseline Code

For each metric calculated, the results showed that the real-life refactorings performed better, as the reductions seen in metrics such as difficulty or time were more significant in the real-life data, showing that were a user to follow the plugin's recommendations, their end result would be harder and take longer to comprehend when compared to what real developers did.

When looking at the maintainability metric, which is an overall metric for the maintainability of the code, taking into account several of the others that were calculated, we can again make the same conclusion as to which refactorings were more effective.

4.5 Refactoring Prediction

The plugin has two major stages of creating the recommendations for the two refactorings we are focusing on: identifying methods/classes that may need refactoring and selecting the code that should be extracted into the new method/class.

We chose to focus on the first step, which is to improve how the plugin identifies problematic methods/classes. The development of the classification model created for such a task is described further below in this section.

4.5.1 Model Development

To create our possible classification models, both for the Extract Method and Extract Class refactorings, we used a pipeline comprised of: data loading and preprocessing, feature selection, and model selection and training, all of which was done in Python.

Data Loading and Preprocessing

Our data was in the form of a SQLite database, which we loaded and proceeded to clean, finding a small amount of infinite data, which we decided to remove instead of replacing with another value, as we have a large amount of data, nearly 25 thousand, allowing us to remove a small amount without endangering our data size.

Aside from that, we have data with values that reach nearly ten thousand, while others barely reach ten, which led us to use a scaler to normalize the data. This scaler was saved locally for further use within the plugin, which we will explore in section 4.5.2.

Feature Selection

We created a covariance matrix of the data, and we chose to remove features that had a high covariance index, which we defined as 0.9 out of 1.0, which resulted in removing 4 features for Extract Method and 2 for Extract Class, as shown in table 4.3.

Metric	**Extract Method**	**Extract Class**
Number of Lines of Code	✓	
Number of Statements	✓	
Halstead Effort	✓	✓
Halstead Length	✓	✓

Table 4.3: Removed Features for each Refactoring Type

The removed metrics are all metrics that use others in their calculation or are used in calculating others, thus resulting in the high covariance values obtained.

Model Selection & Training

Our data is comprised only of instances where a refactoring occurred, as we cannot assume that a code alteration that wasn't a refactoring was a change where a refactoring wasn't meant to happen.

Thus, we used one-class classification models, which are focused on detecting anomalies in a dataset, present in the Python package scikit-learn [PVG+11]: One-Class SVM, Isolation Forest, and Elliptic Envelope.

We chose One-Class SVM as it is highly effective in high-dimensional datasets, of which ours is, both in the case of Extract Method and Extract Class. Aside from that, it's a memory-efficient algorithm, making it especially useful in the case of the Extract Method due to the larger dataset.

Isolation Forest is non-parametric, meaning it doesn't make assumptions about the data distribution, thus suited for high-dimensional datasets. While it is more suited towards larger datasets, its random partitions also make it an effective algorithm on smaller datasets.

Lastly, Elliptic Envelope is once again effective in high-dimensional datasets.

We chose these methods for their versatility, robustness, and efficiency, allowing us to test and find the best anomaly detection system for each refactoring.

After this selection, we needed to understand the parameters that would result in the best model, so we resorted to hyperparameter tuning on each model to find them, for which the parameters tested and selected can be found in appendix A.

As mentioned before, our data is fully comprised of data where a refactoring occurred, which means that a measure such as precision, which is the ratio of true positives to the sum of true positives and false positives, won't be a good measure to test our model's performance, as the lack of false positives will mean a precision of 1.0 at all times.

Model	**Recall**
One Class SVM	0.904
Isolation Forest	0.897
Elliptic Envelope	**0.989**

Table 4.4: Comparison of the Different Extract Method Models

Model	**Recall**
One Class SVM	0.903
Isolation Forest	0.899
Elliptic Envelope	**0.988**

Table 4.5: Comparison of the Different Extract Class Models

Thus, we chose to focus on the recall measure, which measures the model's ability to identify all relevant instances and, as can be seen by the results in table 4.4 and table 4.5, which detail the results for the Extract Method and Extract Class models, respectively, we were able to obtain high values of recall in all models.

Taking this into account, we obtained the best results for both Extract Method and Extract Class with Elliptic Envelope, followed by One-Class SVM, and, lastly, Isolation Forest. We saved the first two for further use.

4.5.2 Model integration

Having our models created and working, we still had the task of integrating them into the plugin and ensuring that the plugin could both use the models and update them as needed.

Python Integration

Our classification models were created using Python, as in terms of machine learning, it is superior to Java, the language used for the development of the plugin, in terms of ease of use, number of libraries, flexibility, performance and scalability.

However, this created the problem of using Python for any tasks required from the models, whether simply getting a prediction from them or re-training them in the future.

Since it not only requires pure Python, but also the use of certain libraries, the user is required to have Python installed on their machine, which is likely as it is a plugin intended for developer use, and provide its path to our tool via the configuration panel, of which the relevant part is shown in fig. 4.3.

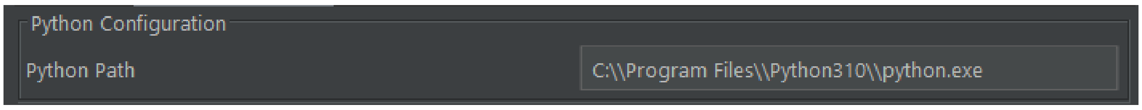

Python Configuration
Python Path C:\\Program Files\\Python310\\python.exe

Figure 4.3: Set Python Path in LiveRef Configuration Panel

Upon getting a path, the plugin will verify its validity and the existence of the required libraries, install them if needed, and use this Python path later on to run the scripts required to do any of the model operations.

Prediction

For the plugin to use the model in its predictions, we replaced the existing logic when it came to the selection of either methods or classes for their respective refactorings with a call to the specific classification model.

Algorithm 1 Get Extractable Fragments for Extract Method from Source File

```
 1: function GETEXTRACTABLEFRAGMENTS(sourceFile)
 2:     fragments $\leftarrow$ empty list
 3:     for each metrics in Values.before.methodMetrics do
 4:         if sourceFile.name does not contain metrics.methodName then
 5:             if metrics.method.body is not null then
 6:                 if PredictionModel.predictEM(metrics, editor.project) then
 7:                     statements $\leftarrow$ metrics.method.body.statements
 8:                     for each statement in statements do
 9:                         fragments.add(getFragmentsFromStatement(statement, metrics))
10:                     end for
11:                 end if
12:             end if
13:         end if
14:     end for
15:     return fragments
16: end function
```

As can be seen in algorithm 1 in line 6, the plugin will now use the created model where, depending on its decision as to whether the method should or should not be the target of an Extract Method, it will continue or not to select the specific code fragments that should be extracted. The Extract Class process is the exact same, barring a selection of methods to extract for the new class instead of code fragments.

Updating

Aside from that, to ensure our model's continuous learning of its user, whenever someone was to perform an Extract Method or Extract Class refactoring utilizing the plugin, we collect similar data, in terms of metrics, to the data present in our training dataset and re-train the model based on it.

However, retraining a model isn't an instant process, and this plugin is a live refactoring tool, thus delays where the plugin is unavailable or the UI is frozen are to be avoided at all costs, which led to us adding a delay between updates in terms of the number of refactorings.

This means that only after an arbitrary number of times, defined by us as 10 but customizable to the user in the configuration panel, will the models be updated. Unfortunately, there will still be a small timeframe of about 30 seconds where the plugin won't be available, but it will be something that happens on a non-regular basis.

4.6 Model Bias

When the plugin is first used, it will have been given extensive data that, while obtained from real scenarios, isn't attuned to the user, meaning it won't take into account the user's preferences.

A tool that someone can use and is able to learn from them is significantly better than one that remains the same, and while this process will happen to our plugin naturally as time progresses, the large number of starting data may make it take too long.

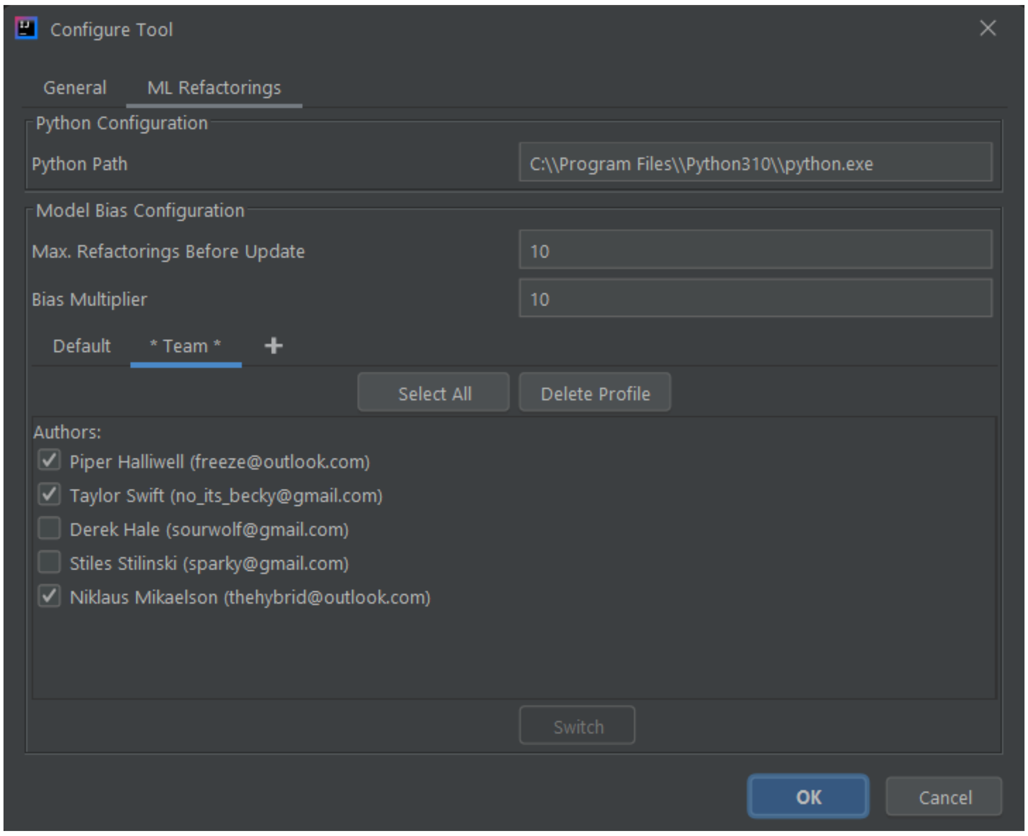
So, in order to make this process quicker and to also make the model even more influenced towards the user, we employed the use of sample weights during the model training and during future re-trainings.

Sample weights can be used in classification models to deal with bias and improve accuracy by assigning different weights to data samples depending on their importance [RDQHY15]. Thus, even though they are often used to counteract bias, we are using them to add bias to our model.

By using the authors present in our database, either through the regular use of the tool or other additions, more on this later on in section 4.7, we can assign the rows of data, which map to refactoring operations, done by the selected authors a higher importance than any other, starting with our default of 10 times the importance, though it's a value that the user can change using the plugin's configuration panel.

Furthermore, the user may not necessarily be coding a project by themselves, as developers often work in teams, so it would be useful that the tool learned not only the user's behaviour but their entire team's.

To achieve this, we allow the user to create different profiles, each of which is comprised of a select number of authors, for which different classification models will be created and, depending on the selected profile, used for the prediction. These profiles are both created and customizable through the configuration panel, shown in fig. 4.4, allowing the user to update them as needed.



Configure Tool

General | ML Refactorings

Python Configuration

Python Path: C:\\Program Files\\Python310\\python.exe

Model Bias Configuration

Max. Refactorings Before Update: 10

Bias Multiplier: 10

Default | * Team * | +

Select All | Delete Profile

Authors:

- [✓] Piper Halliwell (freeze@outlook.com)
- [✓] Taylor Swift (no_its_becky@gmail.com)
- [ ] Derek Hale (sourwolf@gmail.com)
- [ ] Stiles Stilinski (sparky@gmail.com)
- [✓] Niklaus Mikaelson (thehybrid@outlook.com)

Switch

OK | Cancel

Figure 4.4: Configuration Panel

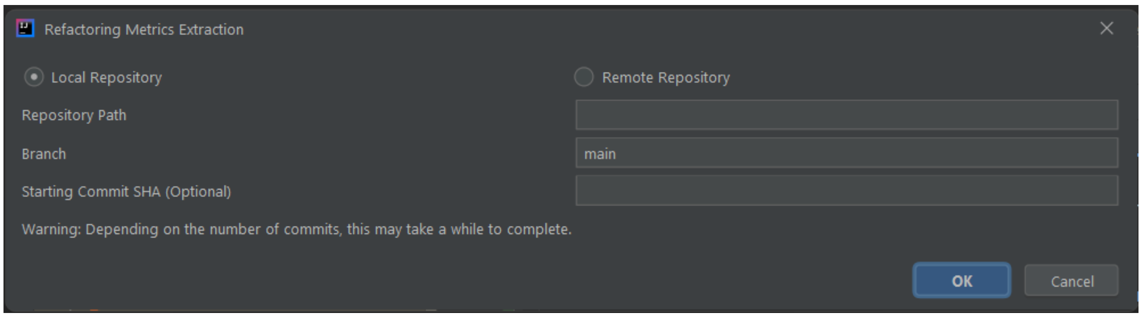
4.7 Repository Metrics Extraction

Even though the model will be continuously updated and increasingly adapted to its users, it would be better if it could be predisposed to them without needing to use it extensively first.

To do so, we created a script that takes a Git repository from a user, as shown in fig. 4.5, and identifies refactorings that had been performed using RefactoringMiner [TME+18]. In order for RefactoringMiner to be compatible with the plugin, we were unable to use the most recent version, which leads to us only supporting this function for the Extract Method refactoring.

The identification of the refactorings is a lengthy process, as it requires traversing through a repository's commits and can lead to a large amount of time needed to execute, especially when it comes to large repositories, though we provide a warning for the user to be aware of this fact, which can be seen at the bottom in fig. 4.5.

After we are able to identify all the refactorings, we extract the metrics we previously mentioned, and add them to our database. In this step, we also collect the authors of each refactoring to add to the tool, allowing for the user to bias the model taking others into account, and move to update all the existing models with the new data.


Refactoring Metrics Extraction

(●) Local Repository ( ) Remote Repository

Repository Path	
Branch	main
Starting Commit SHA (Optional)	

Warning: Depending on the number of commits, this may take a while to complete.

Figure 4.5: Pop-up for Repository Metrics Extraction

Chapter 5

Validation

With this thesis, we intended to improve on an existing refactoring recommendation tool to provide better suggestions. To understand whether we did so or not and answer our research questions, we had to validate our tool, which we did by performing an automated analysis, further described in this section.

5.1 Methodology

The method we used was automated analysis through the use of our plugin with real-life data, being able to compare not only the results obtained with that very same data but with the plugin before we'd made any alterations, as we'd collected that data before, which can be seen in section 4.4.

For this, we selected real-life data for both the Extract Method and Extract Class refactorings and applied our plugin to a method or class before they were refactored to understand whether or not the plugin would suggest any refactorings at all.

Comparing these results with the same process, over the exact same data, with the original plugin allowed us to understand whether or not our changes have resulted in more refactoring suggestions aligned with real developers' thoughts.

5.2 Results

We ran our plugin and saved the for each of the refactoring types, further detailing it in this section.

5.2.1 Extract Method

Previously, in section 4.4 we had calculated the amount of refactoring opportunities found by the original plugin, in case of Extract Method, in real-life data where developers had performed the refactoring.

As can be seen in table 5.1, we repeated the process with the same data, now for our method, testing with the two best classification models, where we were able to obtain a higher percentage of refactoring opportunities found in both models, though the Elliptic Envelope excelled by finding around 13% more opportunities than the original method did.

Method	**Percentage of Opportunities**
Original Plugin	55.98%
One-Class SVM	60.43%
Elliptic Envelope	69.08%

Table 5.1: Comparison of Percentage of refactoring opportunities found for Extract Method

5.2.2 Extract Class

For the Extract Class we hadn't run the data with the original plugin, so we needed a few extra steps that included extracting an extra 400 rows of data, different from the data used to build and train the classification models, where we would be able to validate our solution.

Our results, shown in table 5.2, detail a very small amount of refactoring opportunities found by the original plugin, at only 5.75%, which we were able to improve to 17.12% and 22% with our One-Class SVM and Elliptic Envelope models, respectively.

Method	**Percentage of Opportunities**
Original Plugin	5.75%
One-Class SVM	17.12%
Elliptic Envelope	22%

Table 5.2: Comparison of Percentage of refactoring opportunities found for Extract Class

5.3 Threats to Validity

Even though we have reached certain conclusions, it is imperative to identify possible threats that could influence the validation process. Thus, we will describe possible threats we encountered and the impacts they may have had.

Low Statistical Power While we were able to collect a large amount of data for the Extract Method refactoring, the Extract Class refactoring only had around 2500 different refactoring instances to build the classification model, which may not be enough to find overall patterns for the various contexts in which Extract Class may happen. While our data being from several different software repositories, thus various developers, may help mitigate this issue, there's still the possibility of drawing biased conclusions.

Data Selection Bias The data we collected was used to create the classification models and test their performance. While we made sure not to have a direct intersection of the data used in both situations, it was still collected within the same context. It could potentially introduce bias to the model with data collected within the same context or with the same tools.

Measurement Bias The data collected relied on two specific tools that defined what an Extract Method and Extract Class refactorings are. It could possibly lead to bias as the plugin may be overly aligned with the conditions set by these tools. While it may be lessened by being two tools, their inherent biases or limitations may have been transferred over to the plugin and be reflected in its performance.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

Refactoring is a crucial process for the maintainability of software systems, especially as their complexity keeps increasing, though its traditional process leaves much to be desired. Refactoring recommendation tools are a way to improve upon this process' efficiency, though its recommendations, while good on paper, may not be the best in a real-world context.

Our solution comes in the form of a classification model that relies on the data of actual developers, thus tuning its recommendations to what the developers truly do, not to mention that it can continuously learn and adapt to its user's preferences, personalising itself over time.

We considered the following hypothesis:

"A refactoring recommendation system based on a dynamic classification model, built with real-life data, will lead to more accurate suggestions when compared to threshold-based methods."

For this, we devised two research questions that we would answer in order to validate it:

RQ1 *"Do threshold-based refactoring recommendations not provide context-aware recommendations?"*
Based on the results we obtained by testing the original plugin, which uses the threshold method, we found that for both the Extract Method and Extract Class refactorings, the number of refactoring opportunities found by the plugin was quite low, especially in the Extract Class refactoring, and that, in the Extract Method refactoring, the recommendations provided by the plugin were of lower quality than the refactorings real developers performed.

RQ2 *"Is a classification model based on real data able to improve on the refactoring recommendations when compared to a threshold method?"*
Through our validation efforts, we were able to see an improvement in the plugin's ability to select the methods or classes that should be the target of an Extract Method or Extract Class

refactoring, respectively, being able to lower the amount of refactoring suggestions missed by the original plugin when compared to the real-life data.

6.2 Main Contributions

Over the course of our work, we focused on the development of the classification model, through its multiple stages that stemmed from gathering data to its integration into the plugin, which allowed us to contribute to the software engineering field in a multitude of ways:

Literature Review A crucial initial step we took was to analyse the existing literature to understand how to approach our problem, thus delving mostly into the multiple ways refactoring recommendation systems are able to provide their suggestions, while also understanding how to develop our solution, thus compiling the many forms of how to identify and gather refactoring data.

Plugin The main focus of our work was on the improvement of the original plugin, which we believe we were able to, which resulted in a tool that can be used by developers during their day-to-day work to improve their refactoring efforts by providing live refactoring recommendations.

Refactoring Data In order to build our classification model, we had to extract a rather large amount of data, thus creating a database which contains diverse code quality metrics before and after an Extract Method or Extract Class refactoring was performed. This database could be expanded and used in the future for other related works.

6.3 Future Work

While we were able to achieve our main goal in this thesis, there is always work that can be done to possibly improve upon our efforts, to extend our hypothesis range, or simply to understand our impact on the current state-of-the-art better.

Adding Refactoring Types We focused on the Extract Method and Extract Class refactorings, as they allowed us to validate our hypothesis, which, now confirmed, could be extended to other refactoring types.

Profile Synchronization Our tool supports the creation of profiles, which can be for an individual user, team, or organization, though, on a single device, the plugin will most likely only be used by one developer, thus only taking into account older data from the team that has been previously provided. Possible profile synchronization could be added to allow for the entire team to contribute to the same model in real-time, making their entire recommendations the same.

Diversifying the Data One of the issues we possibly have is that our dataset was obtained by only two tools, which may have resulted in those tool's biases being transferred into our plugin. One way to mitigate this issue is to improve upon the training dataset with data extracted from other refactoring identification tools.

Extended Testing While our changes to the plugin didn't significantly impact the plugin's usage, only adding new options to the configuration menu, its recommendations have changed. Testing in a real development environment over a significant period of time might result in possible issues being brought to light and lead us to a deeper understanding of whether the plugin can truly adapt to a developer's mindset over time.

Throughout our development, we tackled the challenge of enhancing refactoring recommendations based on real-life data. Though we only focused on two refactorings, we achieved successful results and created a knowledge base that can be built on top of, and can be used in the future as a foundation for better context-aware refactoring recommendations.

Bibliography

[Alz22] Musaad Alzahrani. Extract class refactoring based on cohesion and coupling: A greedy approach. *Computers*, 11(8):123, 2022. URL: https://www.mdpi.com/2073-431X/11/8/123.

[AMO] E. AlOmar, M. W. Mkaouer, and A. Ouni. Can refactoring be self-affirmed? an exploratory study on how developers document their refactoring activities in commit messages. In *2019 IEEE/ACM 3rd International Workshop on Refactoring (IWoR)*, pages 51–58. doi:10.1109/IWoR.2019.00017.

[APM+22] Eman Abdullah AlOmar, Anthony Peruma, Mohamed Wiem Mkaouer, Christian D. Newman, and Ali Ouni. An exploratory study on refactoring documentation in issues handling, 2022. doi:10.1145/3524842.3528525.

[BCL+] G. Bavota, B. De Carluccio, A. De Lucia, M. Di Penta, R. Oliveto, and O. Strollo. When does a refactoring induce bugs? an empirical study. In *2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation*, pages 104–113. doi:10.1109/SCAM.2012.20.

[BDLM+] Gabriele Bavota, Andrea De Lucia, Andrian Marcus, Rocco Oliveto, Fabio Palomba, and Michele Tufano. Aries: An eclipse plug-in to support extract class refactoring. https://tufanomichele.com/publications/W1.pdf (accessed July 3, 2024).

[BH] A. Bagheri and P. Hegedűs. Is refactoring always a good egg? exploring the interconnection between bugs and refactorings. In *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*, pages 117–121. doi:10.1145/3524842.3528034.

[CAC+17] S. Charalampidou, A. Ampatzoglou, A. Chatzigeorgiou, A. Gkortzis, and P. Avgeriou. Identifying extract method refactoring opportunities based on functional relevance. *IEEE Transactions on Software Engineering*, 43(10):954–974, 2017. doi:10.1109/TSE.2016.2645572.

[CBK09] Varun Chandola, Arindam Banerjee, and Vipin Kumar. Anomaly detection: A survey. *ACM Comput. Surv.*, 41(3):Article 15, 2009. doi:10.1145/1541880.1541882.

[DAN] A. K. Dipongkor, I. Ahmed, and N. Nahar. Move method recommendation using call frequency of methods and attributes. In *2018 Joint 7th International Conference on Informatics, Electronics Vision (ICIEV) and 2018 2nd International Conference on Imaging, Vision Pattern Recognition (icIVPR)*, pages 76–81. doi:10.1109/ICIEV.2018.8641069.

[DR11] Barthélémy Dagenais and Martin P. Robillard. Recommending adaptive changes for framework evolution. *ACM Trans. Softw. Eng. Methodol.*, 20(4):Article 19, 2011. doi:10.1145/2000799.2000805.

[FAR22] Sara Fernandes, Ademar Aguiar, and André Restivo. A live environment to improve the refactoring experience, 2022. doi:10.1145/3532512.3535222.

[FAR23] Sara Fernandes, Ademar Aguiar, and André Restivo. Liveref: a tool for live refactoring java code, 2023. doi:10.1145/3551349.3559532.

[Fer19] Sara Filipe Couto Fernandes. *Supporting Software Development through Live Metrics Visualization*. Thesis, 2019. doi:https://hdl.handle.net/10216/121655.

[FFYI] Kenji Fujiwara, Kyohei Fushida, Norihiro Yoshida, and Hajimu Iida. Assessing refactoring instances and the maintainability benefits of them from version archives. In Jens Heidrich, Markku Oivo, Andreas Jedlitschka, and Maria Teresa Baldassarre, editors, *Product-Focused Software Process Improvement*, pages 313–323. Springer Berlin Heidelberg. doi:10.1007/978-3-642-39259-7_25.

[FGL12] S. R. Foster, W. G. Griswold, and S. Lerner. Witchdoctor: Ide support for real-time auto-completion of refactorings. In *34th International Conference on Software Engineering (ICSE)*, International Conference on Software Engineering, pages 222–232, 2012. Foster, Stephen R. Griswold, William G. Lerner, Sorin 0270-5257. URL: <GotoISI>://WOS:000312908700021.

[FGLD] L. Franklin, A. Gyori, J. Lahoda, and D. Dig. Lambdaficator: From imperative to functional programming through automated refactoring. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 1287–1290. doi:10.1109/ICSE.2013.6606699.

[FNJ$^+$16] Mário André de F. Farias, Renato Novais, Methanias Colaço Júnior, Luís Paulo da Silva Carvalho, Manoel Mendonça, and Rodrigo Oliveira Spínola. A systematic mapping study on mining software repositories, 2016. doi:10.1145/2851613.2851786.

[Fow] M. Fowler. "CodeSmell". https://martinfowler.com/bliki/CodeSmell.html (accessed Nov. 22, 2023).

[Fow18] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Pearson Education, 2018. URL: https://books.google.pt/books?id=2H1_DwAAQBAJ.

[Hal77] Maurice H. Halstead. *Elements of Software Science (Operating and programming systems series)*. Elsevier Science Inc., 1977.

[Has] A. E. Hassan. The road ahead for mining software repositories. In *2008 Frontiers of Software Maintenance*, pages 48–57. doi:10.1109/FOSM.2008.4659248.

[HPT22] J. Han, J. Pei, and H. Tong. *Data Mining: Concepts and Techniques*. Elsevier Science, 2022. URL: https://books.google.pt/books?id=NR1oEAAAQBAJ.

[HRP$^+$15] Ben Hoyle, Markus Michael Rau, Kerstin Paech, Christopher Bonnett, Stella Seitz, and Jochen Weller. Anomaly detection for machine learning redshifts applied to sdss galaxies. *Monthly Notices of the Royal Astronomical Society*, 452(4):4183–4194, 2015. doi:10.1093/mnras/stv1551.

[HTL$^+$22] Xiaofeng Han, Amjed Tahir, Peng Liang, Steve Counsell, Kelly Blincoe, Bing Li, and Yajing Luo. Code smells detection via modern code review: a study of the openstack and qt communities. *Empirical Software Engineering*, 27(6):127, 2022. doi:10.1007/s10664-022-10178-7.

[KCH] R. Krasniqi and J. Cleland-Huang. Enhancing source code refactoring detection with explanations from commit messages. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 512–516. doi:10.1109/SANER48275.2020.9054816.

[Kuk] Nupul Kukreja. "Measuring Software Maintainability". https://quandarypeak.com/2015/02/measuring-software-maintainability/ (accessed Jun. 12, 2024).

[LTZ] F. T. Liu, K. M. Ting, and Z. H. Zhou. Isolation forest. In *2008 Eighth IEEE International Conference on Data Mining*, pages 413–422. doi:10.1109/ICDM.2008.17.

[MAM$^+$22] Licelot Marmolejos, Eman Abdullah AlOmar, Mohamed Wiem Mkaouer, Christian Newman, and Ali Ouni. On the use of textual feature extraction techniques to support the automated detection of refactoring documentation. *Innovations in Systems and Software Engineering*, 18(2):233–249, 2022. doi:10.1007/s11334-021-00388-5.

[MPSS] Raimund Moser, Witold Pedrycz, Alberto Sillitti, and Giancarlo Succi. A model to identify refactoring effort during maintenance by mining source code repositories. In Andreas Jedlitschka and Outi Salo, editors, *Product-Focused Software Process Improvement*, pages 360–370. Springer Berlin Heidelberg. doi:10.1007/978-3-540-69566-0_29.

[MT04] T. Mens and T. Tourwe. A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139, 2004. doi:10.1109/TSE.2004.1265817.

[NPT] C. Napoli, G. Pappalardo, and E. Tramontana. Using modularity metrics to assist move method refactoring of large systems. In *2013 Seventh International Conference on Complex, Intelligent, and Software Intensive Systems*, pages 529–534. doi:10.1109/CISIS.2013.96.

[OM16] Matteo Orrú and Michele Marchesi. Assessment of approaches for the analysis of refactoring activity on software repositories an empirical study, 2016. doi:10.1145/2962695.2962717.

[Pia] V. Piantadosi. On the evolution of code readability. In *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 597–601. doi:10.1109/ICSME55016.2022.00082.

[PVG+11] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Édouard Duchesnay. Scikit-learn: Machine learning in python. *Journal of Machine Learning Research*, 12(85):2825–2830, 2011. URL: http://jmlr.org/papers/v12/pedregosa11a.html.

[RDQHY15] C. X. Ren, D. A. I. D. Q, X. He, and H. Yan. Sample weighting: An inherent approach for outlier suppressing discriminant analysis. *IEEE Transactions on Knowledge and Data Engineering*, 27(11):3070–3083, 2015. doi:10.1109/TKDE.2015.2448547.

[RSG08] Jacek Ratzinger, Thomas Sigmund, and Harald C. Gall. On the relation of refactorings and software defect prediction, 2008. doi:10.1145/1370750.1370759.

[SA20] Joy Christy Antony Sami and Umamakeswari Arumugam. A vf-imf cohesion metric for object-oriented classes. *Journal of Computer Science*, 16(4), 2020. URL: https://thescipub.com/abstract/jcssp.2020.422.429, doi:10.3844/jcssp.2020.422.429.

[SBPK06] Tobias Sager, Abraham Bernstein, Martin Pinzger, and Christoph Kiefer. Detecting similar java classes using tree algorithms, 2006. doi:10.1145/1137983.1138000.

[SCRT17] Christian Marlon Souza Couto, S C Rocha, Henrique, and Ricardo Terra. Quality-oriented move method refactoring. In *BENEVOL 2017 - 16th BElgian-NEtherlands software eVOLution symposium*, pages 1–5, 2017. Computer Science [cs]/Programming Languages [cs.PL]Conference papers. URL: https://inria.hal.science/hal-01663666.

[SHPCP21] Jaime Sayago-Heredia, Ricardo Pérez-Castillo, and Mario Piattini. A systematic mapping study on analysis of code repositories. *Informatica*, 32(3):619–660, 2021. doi:10.15388/21-INFOR454.

[SP20] Martin Steinhauer and Fabio Palomba. Speeding up the data extraction of machine learning approaches: a distributed framework, 2020. doi:10.1145/3416505.3423562.

[SRA$^+$22] Armando Sousa, Gisele Ribeiro, Guilherme Avelino, Lincoln Rocha, and Ricardo Britto. Sysrepoanalysis: A tool to analyze and identify critical areas of source code repositories, 2022. doi:10.1145/3555228.3555281.

[Ste] Chris Stead. "JS CodeFormer: Javascript Refactoring Code Automation". https://marketplace.visualstudio.com/items?itemName=cmstead.js-codeformer (accessed Jun. 17, 2024).

[SV] D. Silva and M. T. Valente. Refdiff: Detecting refactorings in version histories. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 269–279. doi:10.1109/MSR.2017.14.

[TME$^+$18] Nikolaos Tsantalis, Matin Mansouri, Laleh M. Eshkevari, Davood Mazinanian, and Danny Dig. Accurate and efficient refactoring detection in commit history, 2018. doi:10.1145/3180155.3180206.

[TTH21] Alexander Trautsch, Fabian Trautsch, and Steffen Herbold. Smartshark 2.2 full, 2021. URL: https://doi.org/10.25625/6WAFNG, doi:doi:10.25625/6WAFNG.

[VR18] N. Vijayaraj and Dr. T. Ravi. An advanced cognitive complexity metric for cohesion factor. 2018. URL: https://api.semanticscholar.org/CorpusID:221880992.

[Wix] Wix. "glean". https://marketplace.visualstudio.com/items?itemName=wix.glean (accessed Jun. 17, 2024).

[WNLB21] Fengcai Wen, Csaba Nagy, Michele Lanza, and Gabriele Bavota. Quick remedy commits and their impact on mining software repositories. *Empirical Software Engineering*, 27(1):14, 2021. doi:10.1007/s10664-021-10051-z.

[YNCI] N. Yoshida, S. Numata, E. Choiz, and K. Inoue. Proactive clone recommendation system for extract method refactoring. In *2019 IEEE/ACM 3rd International Workshop on Refactoring (IWoR)*, pages 67–70. doi:10.1109/IWoR.2019.00020.

Appendix A

HyperParameter Tuning

This appendix will detail the hyperparameter tuning process for the 2 models, Extract Method and Extract Class, in terms of the parameters tested and the best obtained.

A.1 One Class SVM

Parameter	**Values**
kernel	rbf, linear, poly, sigmoid
nu	0.1, 0.2, 0.3
gamma	scale, auto, 0.001, 0.01, 0.1, 1, 10
degree	2, 3, 4, 5
coef0	0, 0.1, 0.5, 1

Table A.1: HyperParameter Tuning Grid for One Class SVM

Parameter	**Extract Method**	**Extract Class**
kernel	sigmoid	
nu	0.1	
gamma	scale	auto
degree	3	2
coef0	0.1	0.5

Table A.2: Best Parameter Values for One Class SVM

A.2 Isolation Forest

Parameter	**Extract Method**	**Extract Class**
max_features	1, 5, 10, 14	1, 5, 10, 15, 18
contamination	0.1, 0.2, 0.3	
n_estimators	50, 100, 200, 300, 400, 500	
max_samples	0.1, 0.2, 0.5, 0.7, 1.0	

Table A.3: HyperParameter Tuning Grid for Isolation Forest

Parameter	**Extract Method**	**Extract Class**
max_features	14	18
contamination	0.1	
n_estimators	200	100
max_samples	0.5	0.1

Table A.4: Best Parameter Values for Isolation Forest

A.3 Elliptic Envelope

Parameter	**Values**
contamination	0.01, 0.02, 0.03, 0.04, 0.05
support_fraction	None, 0.7, 0.8, 0.9
assume_centered	True, False

Table A.5: HyperParameter Tuning Grid for Elliptic Envelope

Parameter	**Extract Method**	**Extract Class**
contamination	0.01	
support_fraction	None	
assume_centered	False	True

Table A.6: Best Parameter Values for Elliptic Envelope