

Enhancing Software Quality: Python Code Smell Detection using Machine Learning techniques and Refactoring Long Methods using Extract Method Algorithm

by

Jannatul Ferdoshi

20301193

Shabab Abdullah

20301005

Kazi Zunayed Quader Knobo

20241020

Mohammed Sharraf Uddin

20241018

A thesis submitted to the Department of Computer Science and Engineering
in partial fulfillment of the requirements for the degree of
B.Sc. in Computer Science

Department of Computer Science and Engineering
Brac University
May 2024

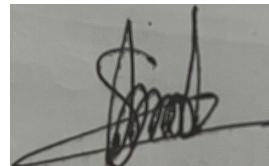
Declaration

It is hereby declared that

1. The thesis submitted is our own original work while completing degree at Brac University.
2. The thesis does not contain material previously published or written by a third party, except where this is appropriately cited through full and accurate referencing.
3. The thesis does not contain material which has been accepted, or submitted, for any other degree or diploma at a university or other institution.
4. We have acknowledged all main sources of help.

Student's Full Name & Signature:

Jannatul Ferdoshi



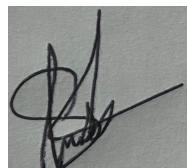
Jannatul Ferdoshi

20301193

Shabab Abdullah

20301005

Zunayed



Kazi Zunayed Quader Knobo

20241020

Mohammed Sharraf Uddin

20241018

Approval

The thesis titled “Enhancing Software Quality: Python Code Smell Detection using Machine Learning techniques and Refactoring Long Methods using Extract Method Algorithm” submitted by

1. Kazi Zunayed Quader Knobo (20241020)
2. Mohammed Sharraf Uddin (20241018)
3. Jannatul Ferdoshi (20301193)
4. Shabab Abdullah (20301005)

Of Spring, 2024 has been accepted as satisfactory in partial fulfillment of the requirement for the degree of B.Sc. in Computer Science on May 21, 2024.

Examining Committee:

Supervisor:



Md. Aquib Azmain
Lecturer
Department of Computer Science and Engineering
Brac University

Thesis Coordinator:
(Member)

Md. Golam Rabiul Alam, PhD
Professor
Department of Computer Science and Engineering
Brac University

Head of Department:
(Chair)

Sadia Hamid Kazi, PhD
Chairperson and Associate Professor
Department of Computer Science and Engineering
Brac University

Ethics Statement

We formally announce the outcomes of our research to be in favor of this concept. This study, in its whole, is devoid of plagiarism. The source material includes citations for any additional information sources. For the purpose of awarding a degree, this thesis has not been submitted, in whole or in part, to any other institution or organization.

Abstract

Python has witnessed substantial growth, establishing itself as one of the world’s most popular programming languages. Its versatile applications span various software and data science projects, empowered by features like classes, method chaining, lambda functions, and list comprehension. However, this flexibility introduces the risk of code smells, diminishing software quality, and complicating maintenance. While extensive research addresses code smells in Java, the Python landscape lacks comprehensive automated solutions. Our paper fills this gap in two ways. Firstly, by constructing a dataset using a tool from existing literature, Pysmell [16]. The tool, given a project directory, determines python files and produces comma separated files for code smells that are present in the python file. We create a dataset containing github projects and run the tool on our dataset. Then we select five comma separated code smell files: Large Class, Long Method, Long Lambda Function, Long Parameter List and Long Message Chain. The comma separated files are then combined to produce a multi-label dataset of code smells. Ensemble techniques and neural networks are trained on the dataset to analyse the performance of machine learning models in predicting code smells given a metric. Secondly, our approach extends to designing and building a simple automated refactoring algorithm, aiming to reduce long method code smells by extracting out large if-else statements and elevate overall software quality. In a landscape where automated detection and refactoring for Python code smells are nascent, our research contributes essential advancements.

Keywords: Python, Code Smells, Code Refactoring, Machine Learning, Code Analysis, Software Quality, Software Maintenance, GitHub Repositories

Dedication

Every challenging endeavor demands our elders, especially the ones closest to our hearts, to put forth personal effort and offer support. In addition to all of the exceptional academics we encountered and learned from while pursuing our bachelor's degrees, and especially our cherished supervisor, Dr. Md. Aquib Azmain, we dedicate our meek efforts to our loving parents, whose love, devotion, motivation, and nightly prayers have made us deserving of this achievement and honor.

Acknowledgement

Firstly, all praise to the Great Allah for whom our thesis have been completed without any major interruption. Secondly, to our supervisor Md. Aquib Azmain sir for his kind support and advice in our work. He helped us whenever we needed help. Finally, to our parents without their throughout support it may not be possible. With their kind support and prayer we are now on the verge of our graduation.

Table of Contents

Declaration	i
Approval	ii
Ethics Statement	iv
Abstract	v
Dedication	vi
Acknowledgment	vii
Table of Contents	viii
List of Figures	xi
1 Introduction	1
1.1 Background	1
1.2 Problem Statement	2
1.3 Research Objective	2
2 Detailed Literature Review	4
2.1 Test Smell Detector for Python	4
2.2 Automated Code Smell Refactoring Tool: FaultBuster	5
2.3 Pysmell: A Metric Based Tool to Detect Code Smells in Python Programs	5
2.4 Code Smell Detection: Towards a Machine Learning-Based Approach	6
2.5 Detecting code smells using machine learning techniques: Are we there yet?	7
2.6 Code smells detection and visualization: A systematic literature review	8
2.7 Understanding metric-based detectable smells in Python software: A comparative study	9
2.8 Comparing and experimenting machine learning techniques for code smell detection	10
2.9 Python code smells detection using conventional machine learning models	10
2.10 Code Smell Detection Using Ensemble Machine Learning Algorithms	11
2.11 Restructuring Programs by Tucking Statements into Functions	12
2.12 Identifying Fragments to Be Extracted from Long Methods	13
2.13 Machine Learning-Based Refactoring Papers: Part I	14

2.14	Machine Learning-Based Refactoring Papers: Part II	16
3	Detailed Literature Review on Compiler Based IDE Plugins	18
3.1	WitchDoctor: IDE Support for Real-Time Auto-Completion of Refactorings	18
3.2	Compiler Driven Code Comments and Refactoring	19
3.3	Large-Scale Automated Refactoring Using ClangMR	20
3.4	cASpER: A Plug-in for Automated Code Smell Detection and Refactoring	20
3.5	A Review-based Comparative Study of Bad Smell Detection Tools	21
3.6	Refactoring Tools and Complementary Techniques	22
3.7	On Experimenting Refactoring Tools to Remove Code Smells	24
3.8	Parallelizing More Loops with Compiler Guided Refactoring.	24
3.9	A Comparative Study on Code Smell Detection Tools	25
3.10	Ten Years of JDeodorant: Lessons Learned from the Hunt for Smells	26
4	Work Plan	28
5	Dataset	29
5.1	Primary Dataset	29
5.2	Secondary Dataset	29
5.3	Tertiary Dataset	30
6	Methodology	31
6.1	A comprehensive comparative analysis between tools that are integrated with compiler and tools that uses machine learning models for refactoring	31
6.1.1	AI-Powered Tools for Refactoring	31
6.1.2	IDE Plug-in Tools For Refactoring	33
6.1.3	Comparative Analysis between IDE plugins /Compilers and Artificial Intelligence to Refactor Code Smells	34
6.2	Code Smells and Metrics	38
6.3	Artificial Neural Network (ANN)	38
6.4	Ensemble Learning using Label Powerset	40
6.4.1	Label Powerset	40
6.4.2	Ensemble Learning Algorithms	40
6.4.3	Model Implementation and their Performance	41
6.5	Refactoring	42
6.5.1	Refactoring of Long Method	42
6.5.2	Long Method and its refactoring solution	43
6.5.3	Detailed Architecture of the Proposed Refactoring Algorithm .	43
7	Result Analysis	54
7.1	Analysis of Code Smell Detection	54
7.1.1	Data Analysis	54
7.1.2	Relationship between input features and the class label	55
7.1.3	Correlation of the code smells and their metrics	56
7.2	Analysis of the Refactoring Algorithm	57
7.2.1	File 1: Basic Example with a a long if-else statement	57

7.2.2	File 2: Example that does not require refactoring	58
7.2.3	File 3: Complex Example - I	59
7.2.4	File 4: Complex Example - II	60
8	Conclusion	63
8.1	Limitations	63
8.1.1	Incomplete Code Refactoring	63
8.1.2	Performance Constraints	63
8.2	Future Work	64
8.3	Conclusion	64
	Bibliography	68

List of Figures

4.1	Flow Chart	28
5.1	Architecture of Pysmell	29
6.1	Evaluation of Machine Learning Models on Refactoring	35
6.2	Architecture of The Neural Network	39
6.3	Label Powerset example	40
6.4	AdaBoost example	41
6.5	XgBoost example	41
6.6	Random Forest example	41
6.7	Flow chart of the naive refactoring algorithm	44
6.8	Structure of the main.py	46
6.9	Output of the dictionary long-condition for the input exmaple	51
7.1	Class distribution of LLF, LM, LMCS, LPL & LC	54
7.2	Relation between Large Class and other Code Smells	55
7.3	Relation between Long Lambda Function and other Code Smells	55
7.4	Relation between Long Method and other Code Smells	55
7.5	Relation between Long Message Chain and other Code Smells	55
7.6	Relation between Long Parameter List and other Code Smells	55
7.7	Correlation Heatmap	56
7.8	File 1 - Before Refactoring	57
7.9	File 1 - After Refactoring	58
7.10	File 2 - Before Refactoring	58
7.11	File 2 - After Refactoring	58
7.12	File 3 - Before Refactoring	59
7.13	File 3 - After Refactoring	60
7.14	File 4 - Before Refactoring	61
7.15	File 4 - After Refactoring	61

Listings

6.1	example_code.py	44
6.2	main.py	46
6.3	example_code.py before applying check_file_format.py	47
6.4	example_code.py after applying check_file_format.py	49
6.5	Output of extract method (<i>new_method.py</i>)	51
6.6	example_code.py after being refactored	53

Chapter 1

Introduction

1.1 Background

The success of any software depends heavily on maintaining code quality in the constantly changing world of software development. Dealing with “code smell” is one of the main obstacles that developers face on this path. A code smell is the term for those subtle and not-so-subtle signs that the codebase may be flawed. It’s similar to that lingering, repulsive smell that signals a problem that needs to be addressed.

Code smell refers to a variety of programming habits and problems that are not technically defects but can impair the maintainability, scalability, and general quality of a codebase[31]. These problems show up as unnecessary code, excessively complicated structures, or poor design decisions. A code smell is a cue for developers to dig deeper into their code, much as a bad smell might indicate underlying issues.

It’s important to identify and fix code smells for a number of reasons. First of all, it has an immediate effect on the development process’s efficacy and efficiency. Code bases with a lot of code smells are more challenging to comprehend, alter, and extend. As a result, it is more likely that while making modifications, genuine problems will be introduced or unwanted side effects will be produced. Additionally, code smell can raise maintenance costs and reduce the agility needed in the quick-paced software development settings of today. Technical debt, which builds up as a result of unchecked code stench, may hinder creativity and impede the advancement of software.

It is evident that code smells can hinder the quality of a software and dealing with such code smells can be frustrating for developers. After researching on automated detection and refactoring of software, we found that many tools have been build to address this issue for JAVA programming language. However, we found a gap in the study for the python programming language. There are many tools which uses IDE plugins and compilers to perform the tasks in question. Nevertheless, we wanted to take an approach that is more future proof by introducing machine learning and deep learning to detect code and refactor code smells in python. Our paper makes the following contributions:

- Makes a comparative analysis on IDE plugins and compilers with machine learning and deep learning models to address the issue.

- Create a dataset with existing tool in the literature called Pysmell and run ANN and ensemble learning models on the dataset to analyse their performance. Moreover, we also analyse the relationship between metrics of code smells.
- Lastly, we propose a simple algorithm to refactor long method code smell using extract method algorithm and analyse its limitations.

1.2 Problem Statement

While building software, it is important to ensure that it is maintained, performs well, and has high longevity. This can be done through the identification and remediation of code smells in software. Despite code smell detection and refactoring being some of the fundamental factors for writing clean code, however, there is a challenge in optimizing and automating this whole procedure. In light of this matter, the main goal of this study is to leverage machine learning techniques to build a systematic, automated, and precise system for figuring out code smells in the Python programming language and fixing them with the aid of refactoring paradigms. With this goal in mind, this study answers the following research questions:

- RQ1: Do machine learning and deep learning techniques perform better than IDE plugins / compilers when it comes to detecting and refactoring codes?
- RQ2: How to design a Python dataset that would allow machine learning algorithms to effectively analyze and recognize code smells?
- RQ3: How to train the machine learning algorithms in order to detect code smells from the Python dataset?
- RQ4: How to create an intelligent system that would provide actionable refactoring suggestions?

1.3 Research Objective

This paper aims to analyse the effectiveness of machine learning models in detection of code smells and building a simple refactoring tool for Python projects. The dataset that will be created for the models to train on will be taken from GitHub repositories. The python files in the repositories will go through a tool that will identify code smells present and create a comma separated file for each code smell. The models will then be trained on the aggregation of the comma separated files to predict the following code smells:

- Large Class.
- Long Method.
- Long Message Chain.
- Long Parameter List.
- Long Lambda Function.

Furthermore, the paper will discuss how a refactoring algorithm was implemented to refactor one of the most prominent code smell, long method, without human intervention, improving the quality of the code base.

We will then evaluate the effectiveness and efficiency of the proposed algorithm by running it on four different python files and finding out the limitations.

Chapter 2

Detailed Literature Review

2.1 Test Smell Detector for Python

The paper [36] begins by outlining the idea of “code smells” before gliding into the topic of “test smells”, which are essentially “code smells” resulting from subparly constructed test cases that have a negative effect on the production code. The writers, Goluben et al., figured that there was a substantial gap in the testing tool scenario by indicating that while test-smell detecting tools are hugely available for JAVA and Scala, they are noticeably absent for Python. The paper [36] talks about a novel tool called Pynose, which was created to fill out this gap. The first steps in their methodology included selecting specific test smells for analysis [36]. Goluben et al. was able to identify 33 different test smells in JAVA, Scala, and Android systems by performing a systematic mapping study of test smells. From the 33 identified smells 17 test smells were customized for Python’s Unittest testing framework after performing a careful filtering process. The paper [36] also emphasizes how crucial it is for the framework of the tool to address Python-specific test smells.

Thereafter, the authors of the paper [36] started the creation of primary dataset, diligently gathering 450 project from GHTorrent, all meeting rigorous criteria: at least 10 contributors, no forks, 1000 commits, and a minimum criteria of 50 stars. The authors made use of a tool, PYTHONCHANGEMINER, that can track changes made to test files to spot Python-specific test smells within this dataset. Completing this step made the authors realize that there was a misuse of assert functions, leading to a specific test smell known as suboptimal assert. Moreover, Pynose’s precision and accuracy in detecting test smells was evaluated using a secondary dataset, comprising of 239 projects.

Finally, Golubev et al. architected and built the Pynose tool in such a clever way so that it can be integrated as a plugin for PyCharm. Firstly, the tool parses and analyses the Python source code using JetBrains’ IntelliJ Platform’s Python Structure Interface (PSI), ensuring both syntactic and semantic examination [36]. After this, the tool selectively extracts classes belonging to unittest.TestCase. Lastly, the tool employs specific detector classes to detect test smells within the extracted classes. The tool intelligently presents the detected code smells in the integrated development environment (IDE) or saves them in a structured JSON format. Remarkably, the evaluation of the tool using the secondary dataset demonstrated an impressive

precision rate of 94% and an equally commendable recall rate of 95.8%, reaffirming its efficacy in detecting and addressing test smells within Python codebases [36].

2.2 Automated Code Smell Refactoring Tool: Fault-Buster

Nagy et al. assert that the process of refactoring presents a formidable and intricate challenge. During the course of refactoring code, developers may inadvertently introduce new defects. The problem is made worse by the widespread belief among tools on the market that developers are naturally skilled at refactoring, a belief that is not always accurate [15]. Furthermore, the research paper underscores that refactoring recommendations constitute the most frequently inquired-about topic on the Stack Overflow platform.

Nagy et al. clarify that the FaultBuster tool has been meticulously engineered to cater to the needs of developers and quality specialists, with the primary aim of facilitating the seamless integration of continuous refactoring as opposed to the conventional approach of deferring such endeavors until the project's completion.

This study's authors [15] divide their tool into three essential parts: a thorough refactoring framework, necessary IDE plugins, and an independent Java Swing client. Firstly, the refactoring framework undertakes the continuous evaluation of source code quality, identifies instances of code smells, and effectuates remedial adjustments in accordance with an embedded refactoring algorithm. Secondly, the IDE plugin serves the crucial function of facilitating the retrieval of outcomes generated by the refactoring framework and effectuating the application of the said refactoring algorithm. Lastly, the Standalone desktop client serves as the conduit for seamless communication with the Refactoring framework.

Nagy et al. subjected their tool to rigorous assessment within the contexts of six distinct corporate entities, where it was employed for the refactoring of extensive codebases totaling 5 million lines of code. As a result of these efforts, the tool adeptly resolved 11,000 code-related issues. Substantiating its efficacy, FaultBuster underwent exhaustive testing and demonstrated the capacity to proficiently rectify approximately 6,000 instances of code smells [15].

2.3 Pysmell: A Metric Based Tool to Detect Code Smells in Python Programs

Chen et al. recognized the scarcity of research concerning the automated identification of Python code smells, which are known to impede the maintenance and scalability of Python software. The primary objective of this investigation [16] is to identify code smells and provide support for refactoring strategies, ultimately enhancing the software quality of Python programs.

At first, Chen et al, acknowledged that certain standard code smells may not be applicable in the context of Python, thus, they took on the task of classifying code smells relevant to Python. The authors of the paper [16] conducted vigorous review of online resources and reference manuals to find out the characteristics of Python code smells. Some of the prominent Python smells are: Large Class, Long Parameter, Long Method, Long Message Chain, and Long Scope Chaining [16].

Next, Chen et al. created a dataset that consisted of five open-source Python libraries, namely django, ipython, matplotlib, scipy, and numpy, comprising a huge codebase encompassing 626,087 lines of code across 4,592 files.

At last, the authors built a tool, Pysmell, which is designed to detect code smells based on relevant metrics. The tool's architecture has three main components:

- i) The code extractor, which cleverly extracts out the python files from a project that is required for further investigation.
- ii) The Abstract Tree Analyzer constructs a abstract syntax tree from the extracted file and also collects the required metric that is set for a code smell.
- iii) The Smell Detector, which detect code smells based on the metric that was found in the previous component.

To finish their study, they evaluated the Pysmell tool, achieving an impressive precision of around 98% and a mean recall of 100% in detecting code smells within Python systems. However, as the study was conducted purely on metric there are some biases on the results of the evaluation [16].

2.4 Code Smell Detection: Towards a Machine Learning-Based Approach

Francesca Arcelli Fontana, Marco Zanoni, and Alessandro Marino discussed the machine learning techniques, which are applied to identify code smells in software development in the paper [8]. Code smells are patterns of inefficiency that can seriously harm the quality and maintainability of software. To keep codebases readable and long-lasting in software engineering, it is essential to find code smells. This paper addresses the difficulties in detecting code smell and suggests a novel approach that makes use of machine learning to improve accuracy and efficiency in this procedure.

The subjective nature of code smell interpretation causes differences in the outcomes. There are many technologies now in use that concentrate on computing metrics while frequently ignoring crucial contextual factors like the domain, size, and design components of the program under analysis. It is challenging to obtain consistent and reliable results because of inconsistencies in threshold settings and metric usage among instruments.

To get over these problems, the authors propose a code smell detection method based on machine learning. More precise and reliable detection strategies are needed, and they highlight the dearth of research on machine learning techniques in this area.

The paper goes into great detail about the methodology, which consists of data collection, code smell selection, application of detection techniques, manual code smell candidate labeling, and machine learning classifier testing.

The authors have chosen and ranked a number of prominent and frequent code smells according to severity, including God Class, Data Class, Long Method, and Feature Envy. This severity rating, which ranges from “No smell” to “Severe smell”, provides developers with a framework for effectively assigning priorities for code smell management. The manual evaluation process consists of looking at potential code smell candidates and grading their seriousness based on observable code features. The human-generated labeled dataset is then used to train the supervised machine-learning classifiers. The authors research a range of classifiers, including Support Vector Machines, Decision Trees, Random Forest, Naive Bayes, JRip, and boosting algorithms, to ascertain the most effective method for detecting code odors.

2.5 Detecting code smells using machine learning techniques: Are we there yet?

Code smells are indicators of poor design choices that can negatively impact source code quality and maintainability. Code smell detection using machine learning (ML) approaches is investigated by Dario Di Nucci et al.[24]. The authors acknowledge the challenges in identifying code smell and discuss the potential advantages of using ML to these issues. They emphasize how programmers must choose between speed and excellent practices due to the increasing complexity of software systems. Code maintenance becomes more difficult when such tradeoffs result in code smells and other technical debt.

The research sheds light on the drawbacks of the existing code smell detection techniques, including their subjectivity and the need to establish parameters that might influence their effectiveness. The authors propose a solution to these issues: code smell detection using machine learning techniques. Code smells may be automatically detected with machine learning (ML) by utilizing source code data to train classifiers. Particular emphasis is placed on supervised machine learning techniques, in which the existence or strength of code smells in code components is assessed using independent variables, or predictors.

The research of Arcelli Fontana et al., which identified four categories of code smells—Data Class, God Class, Feature Envy, and Long Method—forms the basis of this study. The majority of the classifiers in this initial collection of data achieved accuracy and F-Measure rates above 95%, indicating potential. The authors arrive at the conclusion that code smells may be identified by machine learning (ML) approaches, and that the approach selected may not significantly affect the result.

Dario Di Nucci et al. continue to question the generalizability of the results. They voice concerns regarding the potential impact of the dataset on the greater performance shown by Arcelli Fontana et al. The original work employed an imbalanced

dataset, with instances impacted by a particular type of code smell and non-smelly cases included in each dataset for that particular sort of fragrance. In order to address these issues, the authors perform a repeatable analysis on a different dataset that includes code components affected by different code smells. The revised dataset more closely resembles real-world situations by less equally distributing stinky and non-smelly occurrences. The latest research claims that the dataset employed, not the innate capabilities of ML models, was the reason for the previous study’s surprise success.

2.6 Code smells detection and visualization: A systematic literature review

Another paper by Craig Anslow [33], Jose Pereira dos Reis, Fernando Brito e Abreu, and Glauco de Figueiredo Carneiro, titled “Code Smells Detection and Visualization: A Systematic Literature Review”, conducts a systematic literature review (SLR) on the topic of code smell detection and visualization. This study examines the various tools and approaches used to detect software code smells as well as the extent to which these approaches have benefited from visualization. Code smells, also referred to as “bad odors”, are problems with software design and code that lower program quality and make maintenance more difficult. The lifespan and general well-being of software systems depend heavily on the ability to recognize and address code smells.

The results of the SLR reveal several key findings:

1. Code Smell Detection Approaches: The three techniques with the highest usage rates for finding code smells are search-based (30.1%), metric-based (24.1%), and symptom-based (19.3%). While metric-based techniques depend on software metrics to identify code smells, search-based approaches seek specific patterns or structures in the code. The main goal of symptom-based techniques is to identify code smells from observable symptoms.
2. Programming Languages: The Java programming language is the most often examined (77.1%) in studies that employ open-source tools for code smell detection.
3. Common Code Smells: God Class (51.8%), Feature Envy (33.7%), and Long Method (26.5%) are the code smells that are most commonly investigated. These code smells are typical examples of design and maintainability problems in software systems.
4. Machine Learning (ML): In 35% of the investigations, machine learning methods are used. Code smell detection is aided by a variety of ML approaches, including genetic programming, decision trees, support vector machines (SVM), and association rules.
5. Visualization-Based Approaches: About 80% of the research just addresses code scent detection and doesn’t offer any methods for visualizing the results. However, when visualization is used, a variety of approaches are used, such as

polymetric views, city metaphors, 3D visualization methods, interactive ambient visualization, and graph models.

The SLR offers a number of tasks to help identify code smells. Reducing subjectivity in the identification and categorization of code smells, expanding the range of detected code smells and programming languages supported, and providing databases and oracles to support the validation of code scent detection and visualization techniques are the strategies mentioned above. Consequently, this extensive literature review provides valuable insights into the current status of code smell detection and visualization. It highlights the need for automated detection methods, especially when dealing with complex and large-scale software systems. The study also emphasizes how crucial it is for practitioners to have improved visualization tools so they can recognize and address code smells effectively.

2.7 Understanding metric-based detectable smells in Python software: A comparative study

Zhifei Chen and colleagues delve into code issues related to Python. They point out that Pythons' simple structure and dynamic characteristics have led to a focus on analyzing code issues compared to rigid languages, like Java and C sharp[22]. The main goal of their study is to identify and categorize code issues in Python programs while also examining how they affect software maintainability. The paper outlines ten code problems in Python. Explains a method, for detecting them using different threshold setting approaches; machine learning, experience-based, and statistical analysis.

This study's primary objective is to identify and define Python-specific code smells and analyze their impact on software maintainability. The article discusses ten Python code smells and develops a metric-based method for identifying them using three separate threshold patterns: experience-based, statistics-based, and tuning models. The study employs a compilation of 106 renowned Python projects sourced from GitHub to analyze and evaluate these methods of identification.

The analysis, in the study, focuses on code smells, which are indicators of software design and implementation issues. It highlights the significance of addressing code smells to enhance and simplify applications. Various methods, such as analysis, history-based machine learning, and metric-based machine learning are discussed for identifying code smells. While metric-based algorithms are commonly used and effective determining thresholds can pose a challenge. Representing code smells in Python using variables may complicate matters and impact readability and maintainability. Establishing benchmarks for testing Python code smells can present difficulties in setting constraints. The study employs three detection approaches to pinpoint and assess Python code smells revealing their prevalence and impact, on software components.

2.8 Comparing and experimenting machine learning techniques for code smell detection

Francesca Arcelli Fontana et al. presented research on using machine learning (ML) approaches to detect code smells[13]. For many years, code smells have been a major issue in software quality improvement. Code smells are difficult to detect since various individuals have diverse ideas and solutions, and these challenges do not follow any common principles of solution. So these issues can be resolved if we can identify the code smells. This research investigates how machine learning (ML) methods may be used to identify code smells automatically.

This article discusses the difficulty of identifying code smells, with an emphasis on interpretation issues and the absence of clear criteria or measurements. It necessitates a less arbitrary and more focused approach to detecting code smells. Implementing machine learning technology would enable monitors to gain knowledge from particular cases, as suggested. The study's primary contribution consists of practical experiments conducted across a range of software systems and machine-learning techniques.

The methodology involves considering specific code smells such as Data Class, God Class, Feature Envy, and Long Method as variables, with independent variables comprising software design metrics. The paper underscores the importance of selecting and labeling example instances based on the results of existing detection tools to ensure consistent and guided data preparation. The findings presented illustrate the performance of all tested ML algorithms on validation datasets. Notably, J48 and Random Forest outperformed other algorithms in terms of performance, while support vector machines exhibited lower results. The paper acknowledges that imbalanced data influences algorithm performance due to the prevalence of code smells. However, the research findings suggest that machine learning methods can achieve a high level of accuracy in identifying code smells. Moreover, it is noteworthy that even with a limited number of training examples, the accuracy rate reached 95 percent.

2.9 Python code smells detection using conventional machine learning models

Rana S. and Hamoud A. [41] made a clear statement on how identifying code smells at an early stage of software development is crucial to improving the quality of the code. Code smells are impoverished code design practices that negatively affect the quality and maintenance of the code. Most of the research by previous researchers was done on detecting code smells in Java and less on any other programming languages. Thus, their field of study focuses on figuring out code smells in the Python language. They have built their own datasets like source code containing Long method and Long Class code smells. After that, they implemented machine learning algorithms to find the expected code smells and automate the whole process.

Rana S. and Hamoud A.[41] have created their datasets containing Python code

smells. This is done because Python is the most used language in creating data science and machine learning models. Therefore, they have designed datasets based on two criteria. Firstly, they have extracted code smells based on class level and method level. Secondly, they have extracted those code smells that are most present in the Java programming language. As a result, they have chosen Long Methods and Large Class code smells for their Python datasets. In the creation of the Python code smell dataset, four Python libraries—Numpy, Django, Matplotlib, and Scipy have been used by the researchers. The resultant dataset consisted of 18 different sets of features, which were categorized as smelly and non-smelly for each code smell. The datasets were further validated through the use of verified tools like Radon, which ensured the quality of code metrics. To ensure machine learning models' high performance in detection, two data pre-processing steps—feature scaling and feature selection—were carried out on the datasets. Later, to detect code smells in the source code, six machine learning algorithms like support vector machines (SVM), random forest (RF), stochastic gradient descent (SGD), decision trees (DT), multi-layer perceptron (MLP), and logistic regression (LR) were applied. Furthermore, to assess the performance of the machine learning models, two different performance calculation measurements were taken into consideration: the Matthews correlation coefficient (MCC) and accuracy.

The decision tree algorithm surpassed all the other algorithms in finding out Long Method code smell with an accuracy of about 95.9% and an MCC score of about 0.90. On the other hand, Random Forest was the best in recognizing Large Class code smells, with an accuracy of 92.7% and an MCC result of about 0.77. However, it was quite strenuous to recognize the Large class code smell more than the Long method code smell.

2.10 Code Smell Detection Using Ensemble Machine Learning Algorithms

Seema et al. [37] discuss that code smells in software can be detected using ensemble machine learning and deep learning algorithms. According to them, previous researchers did not consider the effects of various parts of metrics on accuracy while finding out the code smells. However, Seema et al. [37] fulfilled this requirement through their research, as they considered all the subsets of metrics, applied the algorithms to each group of metrics, and found their effects on the model's performance and accuracy. Seema et al. [37] expose code smells by building a model. At first, they figured out the datasets of code smells and applied min-max normalization to scale features. Then, the SMOTE class balancing procedure is applied and on the resultant dataset, Chi-Square FSA is implemented to extract the best features. Later, the datasets underwent several ensemble ML techniques, and to improve the performance of the algorithms, cross-fold validation was done. Lastly, different kinds of performance calculations like F-measure, sensitivity, Cohen Kappa score, AUC ROC score, PPV, MCC, and accuracy were calculated to examine the model's performance.

Four types of code smell, like God Class, Data Class, Long Method, and Feature

Envy in the Java programming language, were taken into consideration for detecting code smells. The class-level datasets were God Class and Data Class and on the other hand, the method-level datasets were Feature Envy and Long Method. To rescale the feature values of the datasets between 0 and 1, a min-max normalization process was carried out. Every collection of each dataset was balanced using the SMOTE technique. It is done to enhance oversampling at random. Pre-processing measures like the Chi-square-based feature selection approach are used to find the finest metrics to build the ensemble machine learning models. In the categorical dataset, Chi-square FSA is typically used. Chi-square examines the relationship between features to assist in choosing the best ones. After pre-processing the datasets, Seema et al. applied five ensemble machine learning algorithms such as AdaBoost, Bagging, Max Voting, Gradient Boosting, and XGBoosting, and two deep learning algorithms, Artificial Neural Networks and Convolutional Neural Networks, to the datasets. In detecting the code smells of each type, all of the algorithms competed with each other to be the most accurate. At first, for detecting data class smell XGboost was the most accurate, with 99.80% accuracy. Secondly, all five ensemble learning techniques were the most accurate, with an accuracy of 97.62% in detecting the God class code smell. Moving on, AdaBoost, Bagging, and XGBoost had an excellent accuracy of 100% in detecting Feature Envy. Lastly, AdaBoost, Gradient Boosting, and XGBoost also had a remarkable accuracy of 100% in Long Method smell detection.

In the second half of their project, Seema et al. [37] computed performance measures to compare the performance of machine learning techniques. The performance measurements are as follows, the number of instances of code smell that machine learning techniques correctly identify is measured by positive predictive value (PPV). Sensitivity gauges how frequently machine learning techniques identify instances of code smell. Positive predictive value (PPV) and sensitivity are measured harmonically by the F-measure, which represents a balance between their values. Based on the percentage of correct and incorrect classifications, the AUC ROC score is used to evaluate the effectiveness of a classification model. In this way, Seema et al were able to examine code smells in software.

2.11 Restructuring Programs by Tucking Statements into Functions

In order to restructure programs by breaking up huge functions into smaller ones, a transformation known as tuck is presented in the paper "Restructuring programs by tucking statements into functions" by Arun Lakhota and Jean-Christophe Depréz[1]. Program readability and maintainability are intended to be enhanced by this transformation, which modifies a program's underlying structure without affecting its functionality. The Wedge, Split, and Fold are the three primary phases in the tuck metamorphosis.

In the Wedge phase, a selection of statements from a slice that include linked calculations and have the potential to build a meaningful function are chosen. To build a new function, this subset of statements is then extracted. The Split step allows

functions to be created in a more organized way by splitting off interleaved calculations through the duplication of code segments. Lastly, the program is restructured by calling the newly formed function in place of the extracted subset of statements in the Fold phase.

This restructuring technique's requirement to handle the complexity that develops when programs change over time and lead to deteriorating structures is what drives it. The tuck transformation reduces program complexity and makes maintenance and future upgrades easier by dividing big functions into smaller, more coherent parts.

Within the framework of similar studies, the study examines the work of other scholars, including Notkin, Bowdidge, Sneed, Griswold, Kang and Bieman, and Kim et al. Additionally, by concentrating on coherence and dividing connected sections of code into distinct functions, these academics have investigated restructuring strategies. But by tucking statements inside functions, the tuck transformation (Lakhotia and Deprez)[1] presents a novel way to restructure programs.

The study also emphasizes the possible uses of automated restructuring methods that use the tuck transformation. By automatically detecting code portions that need to be restructured and carrying out the appropriate modifications, these tools can aid in lessening the degradation of a program's structure. Alternatively, programmers can choose and perform the restructure procedures during ordinary code revisions by interactively integrating the tuck transformation into interactive development environments.

To sum up, Lakhotia and Deprez's [1]tuck transformation offers a useful technique for reorganizing programs by breaking down big routines into smaller, easier-to-manage components. This transformation advances the general objective of software reengineering and restructuring by tackling the complexity and maintainability challenges associated with developing software systems. The methodology described in this work provides useful insights on program restructuring strategies that might improve the robustness and lifespan of software systems.

2.12 Identifying Fragments to Be Extracted from Long Methods

Enhancing software quality, maintainability, and extensibility is largely dependent on refactoring, and software maintenance and evolution are essential components of software development. Long and complicated procedures that are difficult to comprehend, maintain, and expand are a prevalent problem in software development. Long procedures can provide difficult-to-understand and alter code, which hinders developers' ability to work effectively.

Long methods became a concern, therefore the idea of "bad smells" in code was invented, with one of the most often found bad smells being "long methods." The idea of keeping functions focused and brief is broken by long methods, which makes

it more difficult for developers to understand the code's intention and how it accomplishes its goals. The software system's progress and upkeep may be hampered by its complexity.

It has been suggested that lengthy procedures be divided into small, more manageable chunks using refactoring approaches like the Extract Method. Although certain code fragments may be automatically removed by current refactoring tools, it still needs human interaction to decide which portions of a lengthy procedure should be extracted. There are inefficiencies in the refactoring process as a result of this laborious and prone-to-error manual method.

To address these issues, academics have put forth a strategy to suggest extractable segments from lengthy procedures, to automate and streamline the refactoring procedure. With the use of a prototype tool named AutoMeD, the methodology helps developers find extractable pieces within lengthy procedures, which lowers the cost of rewriting difficult and lengthy code.

An evaluation of the suggested method on a challenging open-source project shows encouraging reductions of over 40% in refactoring expenses. AutoMeD provides a workable solution to the issue of lengthy procedures in software development, enhancing code readability, maintainability, and overall program quality by automating the detection of extractable pieces. In addition, the approach's accuracy, effect on refactoring costs, and influence on software quality have been the main areas of attention for the evaluation. By looking into these areas, researchers want to present empirical proof of how well the suggested strategy works to solve the problems that lengthy procedures in software systems present.

The work on extracting recommended parts from lengthy procedures is a significant addition to the field of software reorganization and maintenance[4]. Developers may improve the quality and maintainability of their software systems and, in turn, create more effective and efficient software development procedures by providing a methodical and automated way to locate and remove code fragments.

2.13 Machine Learning-Based Refactoring Papers: Part I

Managing code quality high in the software industry is very important. Dealing with code smell plays a key part of it. While compiler based IDE showed some effectiveness on detection and refactoring of code smell, machine learning based approaches have become a more efficient approach. This literature review on five papers highlights why machine learning-based refactoring is preferred over compiler-based approaches, particularly for our research on enhancing software quality through Python code smell detection and refactoring long methods using the Extract Method algorithm.

The study on the Effectiveness of Supervised Machine Learning Algorithms in Predicting Software Refactoring by Mauricio Aniche , Erick Maziero et al.[30] describes

the power of machine learning systems for software refactorings. The authors used a dataset of over two million refactorings from 11,149 real-world projects from the Apache, F-Droid, and GitHub ecosystems to train six different machine learning algorithms which are Neural Network, Support Vector Machine, Naive Bayes, Decision Trees, Random Forest, and Logistic Regression. With an accuracy that is frequently greater than 90%, shows how well the models can predict 20 distinct refactoring actions at the class, method, and variable levels. Each algorithm's precision, recall (98% and 87%, respectively) and accuracy metrics are carefully examined to demonstrate how well it can find refactoring possibilities.

The study "A Survey of Deep Learning Based Software Refactoring" [44] emphasizes the deep learning applications in code smell refactoring. The majority of the study focused on the detection of code smell (56.25%), refactoring solution (33.33%) and 6.25% and 4.17% were towards the end-to-end code transformations refactoring and the mining of refactorings. The study also shows deep learning techniques including CNN, RNN, and GNN and notes their contributions in various tasks. And also manual constructed heuristics is needed in traditional refactoring where deep learning techniques reduce these requirements.

The study on method-level refactoring prediction in five open-source Java projects emphasizes on refactoring opportunities with machine learning techniques[27]. The study highlights the high accuracy rate of classifiers like RUSBoost and SMOTE with 98.47% . Logistic Regression achieved an accuracy of 98%. Naive Bayes with an accuracy of 92.4%. BayesNet demonstrated an accuracy of 84.6% and RBFN (Radial Basis Function Network) achieved an accuracy of 99% and Random Forest: Demonstrated an accuracy of 98.8% where AdaBoost and LogitBoost demonstrated an accuracy of 97.8% and 98.2% respectively. ANN+GD (Artificial Neural Network with Gradient Descent) attained an accuracy of 99% and ANN+LM (Artificial Neural Network with Levenberg-Marquardt) with an accuracy of 98.4%.

Parveena Sandrasegaran and Sivakumar Vengusamy's paper, "Enhancing Software Quality Using Artificial Neural Networks to Support Software Refactoring,"[34] focuses on the application of artificial neural networks (ANN). It shows the enhancement of software quality by emphasizing in the identification and terminating of code smells that creates barriers in software development. It also showcases the ANN model's ability to measure software quality characteristics like maintainability, efficiency, and reusability with an 85% accuracy rate. Furthermore, the ANN model detects code smells with an accuracy of 87% from SciTools and metric factors like Cyclomatic Complexity (CC) data. Moreover, 15% performance boost was shown when compared to typical Machine Learning models in software quality metrics prediction. Therefore, the ANN model's ability to effectively and methodically improve software quality is demonstrated by its ability to detect and minimize code smells.

The study of Dimah Al-Fraihat et al. focused on automatic machine learning (AutoML) [43] approaches for detection of feature envy code smells and refactoring using move method. The "WeightedEnsemble L2" model is used in this paper for automated identifying and categorizing feature envy cases in software systems. The

model shows an accuracy gain from 58% to 77% in successfully detecting code smells. The model's high predictive power is further highlighted by its macro average F1-score of 57%.

In conclusion, the advancement of software engineering has emerged with the help of various machine learning techniques. Compared to compiler based refactoring, these techniques offer more accurate, effective, and automated solutions, which increases productivity during development and lead to better code quality and maintainability. These models' adaptability and strong performance in a variety of scenarios and projects demonstrate their usefulness in real-world situations and their capacity to revolutionize software development methodologies.

2.14 Machine Learning-Based Refactoring Papers: Part II

Software refactoring, which aims to improve a program's internal structure without changing its external behavior, is a crucial approach in software engineering. Integrated Development Environments (IDEs) based refactoring techniques most of the time depends on guidelines and heuristics. These techniques are not able to understand the dynamic and complicated software systems. With the help of machine learning based refactoring we can solve these problems. This analysis based on 5 papers highlights the findings and advantages of using machine learning for code smell refactoring. ML has improved the precision, effectiveness, and range of refactoring techniques by working on numerous datasets and efficient algorithms.

The study on a machine learning approach to software model refactoring[40] showcases the detected functional decomposition instances in UML class diagrams with a high accuracy rate of more than 90%. Functional decomposition gave their model an F1 score of 0.90, recall of 0.93, and accuracy of 0.87 when it came to labeling test UML class models as faulty. The algorithm is empirically evaluated and shows high accuracy. Compared to standard refactoring techniques that target specific smells and result in a cycle of quality evaluation, refactoring, and synchronizing the side-effects, the main addition is the innovative perspective on design defects.

The study on "An Empirical Study of Refactorings and Technical Debt in Machine Learning Systems" [35]describes the link between refactorings and technical debt. It describes how refactoring can lead to technical debts in the software industry. The aim of the paper is to guide how to keep a balance between refactoring and technical debt. It shows the consequences of machine learning based refactoring that is related to technical debts. It also provides important insights for managing technical debt in machine learning systems by identifying common refactoring patterns and technical debt categories through the analysis of 26 projects totaling over 4.2 million lines of code.

The study emphasizes on developers' abilities and intuition and the difficulties faced by the software developers are also discussed. The paper[29] used a deep learning technique which is the gated recurrent unit (GRU). The primary goal of this is to

find class-level code that needs refactoring. The dataset of the model was even open-source Java projects. In the experimental study,[29] the impact of data sampling on the model’s performance was investigated and was carried out both before and after the dataset was balanced. The results were remarkable for the prediction of code restructuring performed admirably. The main contribution of the research is an analysis of how well deep learning methods, in particular the GRU, work for developing prediction models for class-level refactoring. It highlights how special it is to do this using the GRU algorithm and the evaluation of the selected Java apps. This study[11] showcases a search-based learning algorithm for software refactoring where the method uses an Artificial Neural Network (ANN) as a fitness function. In comparison to current software restructuring strategies, the methodology suggested in this work has a number of advantages. In terms of precision and recall, it performs better than current methods, to start. While other techniques only achieve 65% and 74% precision, the methodology offers refactorings that are already performed by software development teams to the following version with an average precision of 80%. Additionally, the suggested method shortens CPU times and increases refactoring efficiency. When compared to other search-based methods, such genetic algorithms and interactive genetic algorithms, the methodology performs better. Finally, prospective users of the refactoring tool evaluated the technique and found positive results in terms of producing workable and effective refactorings. These benefits demonstrate the suggested approach’s efficacy and efficiency in comparison to other methods.

The paper[19] emphasizes a machine learning-based approach for code smell severity classification rather than direct code refactoring where conventional IDE refactoring tools provide automatic assistance for implementing pre-established refactoring procedures to enhance the quality and maintainability of code. Inside the IDE environment, developers may use these tools to apply refactoring operations (such as Extract Method, Rename Variable, etc.) to their codebase. The paper[19] used data-driven techniques for the classification of the severity of code smells based on metrics where IDE rely on static analysis to suggest and apply refactoring.

In conclusion, machine learning based code smell refactoring shows a significant improvement in performance and achieved a high accuracy. Machine learning models such as GRU, ANNs have showcased their ability to correctly detect complex code flaws and technical debt in software systems. These models offer automated, efficient, and data-driven approaches that enhance the quality, maintainability, and scalability of software. Developers can achieve more effective refactoring outcomes by using machine learning techniques for code smell detection and refactoring.

Chapter 3

Detailed Literature Review on Compiler Based IDE Plugins

3.1 WitchDoctor: IDE Support for Real-Time Auto-Completion of Refactorings

The WitchDoctor paper[6] presents a system that eliminates the cognitive prerequisites required by programmers to start refactorings and thus attempts to resolve the problem of underused IDE-provided automated refactoring support. It works by observing the programmer’s coding activity in real time, detecting refactoring in progress, and completing it before the programmer can finish. This is to prevent programmers from needing to know a particular refactoring operation to be refactored, the name of the refactoring, and how to apply it from the IDE.

The WitchDoctor paper describes[6] a system that gets rid of the mental preparations that coders need to begin refactorings. This tries to solve the issue of automatic reworking support that isn’t used enough by IDEs. It watches what the programmer is doing with their code in real-time, finds rewriting that is happening, and finishes it before the coder can. This is done so that coders don’t have to remember the name of the change, the action that needs to be refactored, and how to use it from the IDE.

WitchDoctor’s ability to automatically find and complete refactorings in real time is helpful for both new and experienced coders. While writing, beginners can get relevant ideas and help, while experts can work faster without being interrupted by directly getting modification tools. The system’s interface suggests finished refactorings to coders, and they can easily accept or reject these ideas while they code. Programmers of all skill levels may benefit from WitchDoctor’s real-time automatic refactoring identification and completion. Experts may work uninterrupted and more effectively by manually using refactoring tools, while novices can get contextual ideas and advice while coding. Programmers are presented with finished refactorings via the system’s interface, which they can easily accept or reject while coding.

The article[6] highlights the importance of interface design to help effectively share refactoring ideas with programmers. It describes plans for developing prototypes

of beginner and expert programmer-targeted interfaces, as well as conducting more practical evaluation studies of WitchDoctor. The system is based on a selective rollback approach relying on textual differencing, delayed identity of AST node mapping, declarative specification language, pattern matching algorithms, and selective rollback to serve refactoring proposals, efficiently, consistently, and flexibly.

WitchDoctor is a giant step for automated refactoring support, in the path to facilitate refactoring tasks (reduce-time effort consuming) and makes it easier for programmers to accomplish more reliable, useful work. We intend to study user interface designs in more detail, discover real-world refactoring practices, and improve the system based on user feedback and usage scenarios.

3.2 Compiler Driven Code Comments and Refactoring

The paper[5] showcases the difficulties that compilers have in leveraging full optimization potential in the code fragments and raises the fact that programmers are the ones who have to further ensure parallel software development is efficient and correct.

The goal of the toolset described in the paper is to assist programmers in understanding the design margins that they need to comply with to exercise these optimizations and to help them narrow a subset of the barrier binaries for automated treatment. Incorporating the compiler with the IDE and making the code comments, which helps the developer understand and fine-tune the code for multi-core architectures, are also a part of the toolset. Today, the paper is attempting to take an approach that has evolved the role of the compiler from merely a passive optimizer into an active guide, working with programmers to produce better-optimized code.

One of the image processing benchmarks was used to evaluate the efficacy of this complete toolset to enable significant speedups by automatic parallelization compared to traditional compiler methodologies mainly for testing purposes at this stage. The results highlight the power of the toolset for refactoring code for amenable optimization and improved acceleration on multi-core architectures. Additionally, the authors also provide the trajectory of further investigations: supporting more kinds of analysis; providing more refactoring; and prioritizing the code comments to be extracted based on the execution frequency and availability for automated refactoring. We also intend to publish the toolset as open-source software free for all once the prototype implementation is more mature.

We believe the paper is a significant advance for the parallel software development field as it introduces a new marginal methodology that helps programmers optimize their code for multi-core architectures with the help of advanced compiler analysis and refactoring advice. The toolset appears to be building up a strong foundation for improved intelligent code optimization and performance, promising a brighter and more efficient parallel software development methodology.

3.3 Large-Scale Automated Refactoring Using ClangMR

The "Large-Scale Automated Refactoring Using ClangMR" [10] method refactors C++ code quickly and effectively using the Clang compiler framework and the MapReduce parallel processor. This helps with the problems that come with maintaining big codebases. The authors stress how important it is to keep current code up to date so that it can be used for new features, old interfaces can be removed, and technical debt can be managed. The tool was made because it's hard to constantly update large chunks of code in a semantically safe way, especially as codebases get bigger.

The ClangMR system is characterized by its high level of parallelization and semantic awareness. It utilizes the semantic knowledge derived from the C++ abstract syntax tree (AST) of the Clang compiler to make educated judgments during the editing process. ClangMR uses the MapReduce architecture to distribute its analysis across numerous machines at the same time. This allows for efficient processing of sophisticated transformations on large amounts of C++ code in just a few minutes. This approach enables code maintainers to effectively restructure extensive codebases while guaranteeing semantic accuracy.

The study[10] highlights the distinctive characteristics of ClangMR, including its versatility, efficiency, and applicability to industrial use cases, setting it apart from conventional regular-expression-based matching tools and constrained refactoring tools commonly found in integrated development environments like as Eclipse. The authors further analyze the scaling difficulties faced by competing tools, highlighting ClangMR's ability to handle large-scale refactoring assignments. They provide an example of a real-world scenario where ClangMR successfully modifies over 35,000 function call sites across 100 million lines of code. In addition, the article offers valuable information about how ClangMR is used at Google, showcasing its effective use in updating callers of outdated APIs and revitalizing old C++ code to meet current standards. The study showcases the efficacy of ClangMR in managing extensive refactoring jobs, including the capability to iterate transformations and keep up with developing codebases.

In short, the literature review of "Large-Scale Automated Refactoring Using ClangMR" characterizes the relevance of the system in the context of mitigation of problems with maintaining and moderating vast C++ codebases ClangMR combines Clang compiler's ability to process semantics and features of MapReduce to provide a useful and versatile tool for code maintainers to quickly and safely refactor code at scale.

3.4 cASpER: A Plug-in for Automated Code Smell Detection and Refactoring

De Stefano et al. The paper "cASpER: [32]A Plugin for Automated Code Smell Detection and Refactoring" introduces a new way for automated code smells detection and refactoring of a code using the plugin for IntelliJ IDEA. Although the issue

many developers face is how to keep their code quality as high as possible while ticking off all the boxes on their to-do list and this often leads to code smells, which are poor design or implementation choices that can affect your program comprehension, maintainability, and team productivity.

The research[32] examines the prior works in this line of software engineering research, particularly in object-oriented code refactoring as well as the exploitation of both structural and semantic measures to improve software modularization. Inspired by these prior works, the paper provides developers with a tool that not only identifies code smells but also suggests automated refactoring opportunities for better-quality source code.

In this manner, cASpER guides developers on a code smell identification and refactoring adventure by offering them visual and semi-automated support for a wide range of code smells, from feature envy to misplaced classes, blobs, and promiscuous packages. Developed to integrate directly into IntelliJ IDEA, the popular integrated development environment, the plug-in provides a user-friendly interface for code analysis and refactoring suggestions that do not change the external behavior of the source code. The accuracy of the code smell detection and refactoring approaches is key for the tool to be effective, as they were previously presented as well as validated through experiments in other research papers. With that in mind, the goal of cASpER was to remove the manual work involved in improving code quality by automating the refactoring process to encourage developers to address code smells proactively instead of being forced to.

To sum up, cASpER is a valuable addition to the research area of automated software engineering for providing a pragmatic approach of detecting and refactoring code smells interactively and efficiently. The plug-in could have a positive effect on how software is developed and the quality of the code. To further our study, we should look at how developers respond to cASpER’s suggestions and add more code smells and detection methods to the tool.

3.5 A Review-based Comparative Study of Bad Smell Detection Tools

The paper ”A Review-Based Comparative Study” [17]presents a systematic literature review (SLR) as the motivation for the work, which focuses on the detection of bad smells in software systems. The study’s objective is to identify and document the tools used for bad smell detection, which are indicators of potential quality problems in software design and code. The authors utilized a thorough set of inclusion and exclusion criteria to choose pertinent tools and papers for the study, following multiple rounds of refinement and data extraction and conducting a thorough review of the papers.

The SLR protocol followed in the study involved defining research questions, inclusion and exclusion criteria, and search strategies to identify relevant literature. The authors utilized various electronic data sources to gather a comprehensive set

of papers published between 2000 and 2015, focusing on bad smell detection tools in the field of computer science. Through a meticulous screening process, the authors identified 84 bad smell detection tools for further analysis.

A key element of this study was the categorization and systematization of features of the identified tools. One of the limitations faced by the authors was that, concerning a few of the tools, complete information could not be obtained concerning what types of bad smells they detect and how they detect these bad smells. This made it difficult for the authors in some cases but they did their best to collect all of the usable data for their review of the tools. It also contains a comparison results of four bad smell detection tools when applied to the detection of large-class and long-method bad smells. It showed agreement, recall precision rate, and usefulness of each tool giving us an idea of how efficient each tool is in detecting specific bad smells.

For the detection of large-class bad smells: inFusion: detected 0 instances in JUnit, 1 instance in MobileMedia JDeodorant: Detected 88 instances in JUnit, 11 instances in MobileMedia PMD: Detected 12 instances in JUnit, 1 instance in MobileMedia JSpirit: Detected 6 instances in JUnit, 2 instances in MobileMedia.

For the Long Method foul smell detection, JUnit detected 0 instances in Fusion, 2 instances in MobileMedia; 48 instances in JUnit, 12 instances in MobileMedia; JUnit detected 0 instances in PMD, 3 instances in MobileMedia; JUnit detected 0 instances in JSpirit; 3 instances in MobileMedia; and 12 instances in JDeodorant.

The results show that JDeodorant found the most Large Class and Long Method instances in both software systems. This shows that it is good at finding these unpleasant smells. It was also shown that PMD could accurately recognize large classes in the MobileMedia system, reaching 100% accuracy. However, JDeodorant did worse than PMD and JSpirit in both the big class and long method when it came to recall and accuracy.

In summary, this study provides an extensive overview of the tools to identify bad smell detection tools, what bad smell detection tools check, and how well they are good at delivering common real-world software quality problems. Its methodological concentration assures that the results are valid and essential. So it serves students and practitioners as a good reference on software maintenance and quality assurance.

3.6 Refactoring Tools and Complementary Techniques

Refactoring Tools and Complementary Techniques—Automated Software Engineering This paper by Martin Drozdz, Derrick G. Kourie, Bruce W. Watson, and Andrew Boake[2] covers how refactoring plays a crucial role in enhancing software design and maintaining the code's maintainability. The authors note the difficulties when working with poorly designed software, demonstrating how difficult it is to modify code

and add features because of coupling. These folks would go on to claim that only by routinely refactoring enterprise systems can you effectively tackle all of the aforementioned concerns in large systems.

Developers present refactoring as a valuable practice that enhances code design and cleanliness without changing its behavior. The study includes a comprehensive survey of refactoring tools available in popular commercial and open-source Integrated Development Environments (IDEs) such as IntelliJ's IDEA, IBM's Eclipse, and Sun's Netbeans. The authors[2] also discuss the potential of automatic code smell detection through static code analysis to identify targets for refactoring.

The paper's audience has valid concerns related to implementing compiler refactorings as an automatic or mostly automatic process because humans need to ensure that the refactored code is correct. It cites Mens (2004) to emphasize that refactoring without developer intervention is dangerous because automated tools may make the situation even worse by performing unnecessary refactorings that decrease overall code quality. In addition to those, the authors write about the advantages of a frictionless design process for developing software and how refactoring helps to transition from legacy design models to modernized models being flexible to adapt reusable components with low maintenance cost.

"Refactoring Tools and Complementary Techniques" Analysis IntelliJ IDEA:

- Outperforms Eclipse in ease of use and productivity for refactoring.
- Offers around 30 similar refactorings.
- Features nearly 500 static code analysis tools.
- High accuracy in detecting code smells and potential refactoring areas. Eclipse:
- Similar to IDEA in refactoring tool landscape.
- surpasses Eclipse in ease of use and productivity.
- Does not provide specific details on refactoring tool accuracy.

Netbeans:

- Criticized for limited refactoring capabilities.
- Lack of support for essential refactorings raises concerns about the IDE's effectiveness in identifying and addressing code smells.

Overall, the paper offers recommendations on how IntelliJ's IDEA, with its performance and concurrency accuracy from its numerous sets of static code analyses, can be advantageous in refactoring endeavors. Eclipse seems to have a satisfactory performance, unlike Netbeans, which seems to be a bit slow and incorrect, especially in areas of refactoring. According to the information presented in the paper, developers who are in search of effective and accurate methods may conclude that IDEA is indeed a suitable option to explore in terms of the reaction to the refactoring tools.

3.7 On Experimenting Refactoring Tools to Remove Code Smells

The paper “On Experimenting Refactoring Tools to Remove Code Smells” by Francesca Arcelli Fontana, Marco Mangiacavalli, Domenico Pochiero, and Marco Zanoni from the University of Milano Bicocca, Milan, Italy, [14]describes an experiment on refactoring tools on how to remove code smells with the aim of enhancing the source code quality of software projects. The authors designed several experiments normalizing code smells with four refactoring tools and evolved their experiences with such an approach.

Both authors stress the importance of refactoring, which is aimed at modifying the existing code for the sake of making it more effective and safe, as discussed by Fowler. In this context, we view refactoring as an essential activity that helps to deal with code smells and improve the quality of software. However, resistance to change indicated by developers’ failure to refactor code can cause code smells to become long-lasting and disrupt software evolution. The authors, in their study, have used four refactoring tools: Eclipse, IntelliJ IDEA, RefactorIT, and JDeodorant, to identify the efficacy of the tools in eliminating code smells. The team realized that each of the tools had advantages and disadvantages that they needed to consider. While discussing this, they observed that JDeodorant is capable of detecting and recommending correct refactorings of code smells but faced certain challenges when using it. Users have appreciated Eclipse and RefactorIT for their clear menus containing refactoring possibilities, while IntelliJ IDEA offered numerous features regarding the analysis and refactoring of the programs.

The study[14] recognizes the general validity threats in this type of research by conducting experimentation only with a few selected systems and tools. There are several directions for further research: the authors would like to investigate more systems in detail and broaden the choice of refactoring tools that help to eliminate code smells.

In conclusion, the paper offers insights into the use of refactoring tools to eliminate code smells in software development projects. The paper emphasizes the importance of selecting the right tool based on specific criteria and the need for further research in this field. The authors’ study can help developers choose more suitable tools for refactoring and enhancing the quality of their code. In summary, the paper adds to the current literature on software maintenance and refactoring by presenting actionable insights and strategies for software developers and researchers keen on improving software quality by eradicating code smell.

3.8 Parallelizing More Loops with Compiler Guided Refactoring.

“Parallelizing More Loops with Compiler-Guided Refactoring”[7] focuses on the important problem of automatic parallelization in parallel leagues. However, it also emphasizes how challenging and delicate this is, pointing out that there are quite a

number of issues that can impinge or halt Automatic Parallelization in real application. The review brings up a comparison of Intelse and IBM production compilers based on a study that shows that, while total vectorization can be quite high at the collective level, many individual benchmark loops are only partially vectorizable. This disparity puts into picture the difficulties faced by the compilers in tuning the code for concurrent process operation.

As a result, the review stresses the fact that parallel applications unmask loop-level parallelism as opposed to instruction-level parallelism. It notes that current compilers frequently generate reports that contain information on compilation concerns while allowing the programmer to determine particular source code elements that hinder optimization. This absence of feedback calls for far more sophisticated interactive compilation systems, which will be very helpful in helping programmers transform their code in a manner amicable to auto-parallelization.

Using such semi-automatic parallelization methodology jointly with special target optimizations, the review emphasizes that application performance might be boosted dramatically. Through addressing these key points, a basis for the proposal and assessment of the presented interactive compilation feedback system in the paper is the literature review, desiring to assist programmers in harnessing the maximum benefits of loop-level parallelism in their parallel computations.

3.9 A Comparative Study on Code Smell Detection Tools

The paper “A Comparative Study on Code Smell Detection Tools” written by Almas Hamid and his collaborators, lecturers of the University of Sargodha in Pakistan,[9] provides comprehensive information on the existing methods used in identifying code smells in software development. The findings of the study reveal that refactoring is a valuable activity that can be used to increase the readability and maintainability of computer programs. Code smells are those signs or markers that suggest the presence of certain kinds of errors or problems in the code base; with the help of recognizing and eliminating these smells, the working developers tend to enhance the quality of the output of the software.

For example, Steve et al. examined code smells and processed them according to the taxonomy discussed in the relevant literature. Their study also established that there were differences in the level of refactoring that was needed to eliminate various categories of code smell. The study also highlighted the fact that some of the code smells are actually misleading, which means they can give a skewed understanding of the effort needed to perform the refactoring.

In addition, Foutse et al. investigated the correlation between code smells and software change-susceptibility. Their study conducted showed that classes with smell are more likely to be changed compared to classes with no smell. Through the assessment of the DECORE strategy for code smell identification, the study offered

knowledge on how different code smells can influence software evolution processes.

Moreover, Fontana et al. shared a report on code smell detection tools, as well as a specific analysis of the effectiveness of those tools regarding the Gantt Project application. The authors highlighted the need to gain access to the rules of detection as well as thresholds of metrics within code smell detection tools to enhance their capability to identify and remedy flawed structural designs of code.

In conclusion, the papers reviewed above provides the right background needed to understand the location of the current study with regard to research on code smell detection and refactoring. Thus, the paper provides the theoretical background for the further comparison of the scientific results related to the detection of code smells based on the analysis of the works of other authors and presents practical suggestions that may be of interest to professionals in terms of maintenance and improvement of software quality within the field of software engineering.

3.10 Ten Years of JDeodorant: Lessons Learned from the Hunt for Smells

The tool JDeodorant is the subject of the study "Ten Years of JDeodorant: Lessons Learned from the Hunt for Smells" [26]. The authors provide methods for proposing and implementing refactoring. In their reflections, they also discuss how JDeodorant evolved, how to propose and implement refactoring possibilities, and how it has affected software engineering practice and research. The study also explores the field's competitive works, lessons gained, and the value of tools like JDeodorant in enhancing the quality of code and design for maintainable software projects.

JDeodorant [26] focuses on figuring out which refactorings to apply, where in the program to apply them, preserving behavior, implementing the refactorings, evaluating the effect on quality attributes, and preserving consistency between the refactored code and other software artifacts. Unlike other IDE refactoring tools, JDeodorant offers tailored solutions for every code stench, taking into account the unique features of the underlying design or code issue. Compared to IDEs' general approaches, JDeodorant's customized method enables it to provide more focused and efficient refactoring advice. Moreover, JDeodorant incorporates refactoring mechanisms, automated Javadoc comment updating, preconditions to maintain behavior, and algorithms for detecting code smell.

Additionally, JDeodorant may be easily integrated into the development environment where developers work because it is available as an Eclipse plug-in. Developers may now access and implement refactoring ideas right within their familiar IDE environment, which improves ease and usefulness. In addition, the tool's minimum installation and configuration effort facilitates its acceptance by academics, practitioners, educators, and students by further streamlining the user experience. In conclusion, JDeodorant is more effective and superior to traditional IDE refactoring tools in terms of enhancing code quality and streamlining the refactoring process

because of its customized strategies for code smells, extensive support for refactoring activities, integration as an IDE plug-in, and user-friendly interface.

Chapter 4

Work Plan

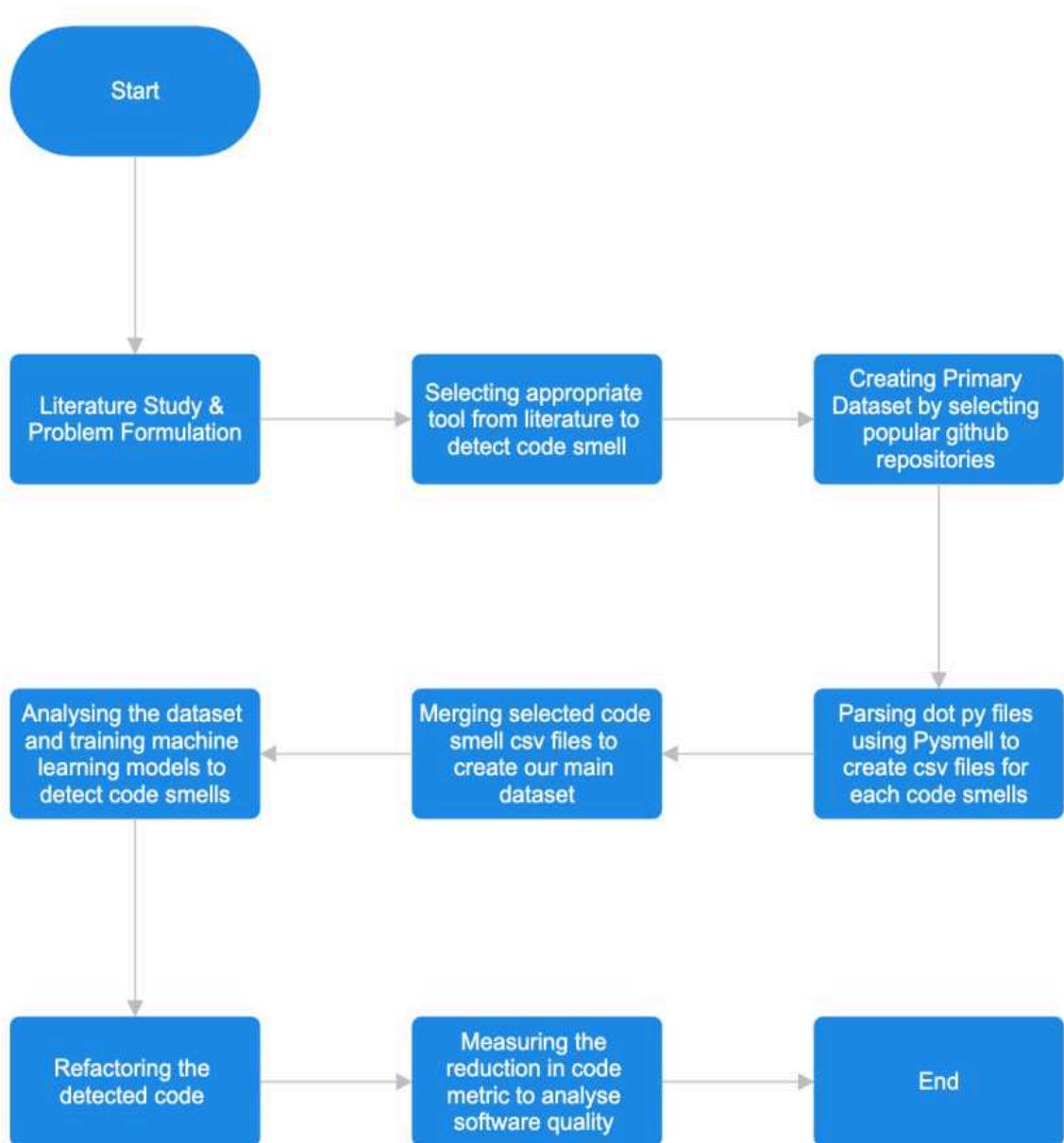


Figure 4.1: Flow Chart

Chapter 5

Dataset

5.1 Primary Dataset

In constructing our initial dataset, we adopted a selective approach by focusing on GitHub project repositories with an impressive following, specifically those garnering more than 1000 stars. This criterion ensured the inclusion of projects widely recognized and embraced by the programming community. Among the noteworthy projects featured in our primary dataset are renowned Python libraries such as Keras, Django, Seaborn, Scipy, and more. These selections were made based on their popularity and significance within the Python programming ecosystem, contributing to a diverse and representative collection.

Ultimately, our primary dataset comprises 50 carefully curated project repositories. Notably, one of these repositories is a project of our own, adding a valuable dimension to our research. This intentional inclusion allows us to explore code smells within the context of our project, providing insights and perspectives that contribute uniquely to the overall findings of our research paper.

5.2 Secondary Dataset

In leveraging the Pysmell tool, as detailed in our literature review, we employed it to construct our secondary dataset. Please refer to the comprehensive literature review (2.3) for an in-depth description of the tool. The architecture of Pysmell is outlined below:

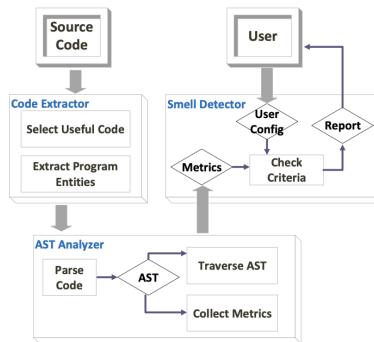


Figure 5.1: Architecture of Pysmell

Upon inputting our primary dataset into the Pysmell tool, it precisely processed the data, generating insightful information. Specifically, it produced a set of comma-separated files for 33 out of the 50 projects in our primary dataset. The tool intricately parsed and analyzed each project, focusing on ten distinct code smells: Large Class (LC), Long Method (LM), Long Message Chain (LMCS), Long Parameter List (LPL), Long Lambda Function (LLF), Long Scope Chaining (LSC), Long Base Class List (LBCL), Long Ternary Conditional Expression (LTCE), Complex List Comprehension (CLC), and Multiply Nested Container (MNC).

For each code-smell category, the Pysmell tool generated seven comma-separated files. These files provided valuable insights, encompassing essential details such as the project name, the names of Python files within the project, the relevant metric, and the instances of code smells associated with the specified metric. This careful breakdown ensures a comprehensive understanding of the code smells identified, enabling us to conduct a thorough and nuanced analysis of our secondary dataset.

5.3 Tertiary Dataset

In the culmination of our research efforts, we curated our final dataset by focusing on five distinct code smells: Large Class (LC), Long Method (LM), Long Message Chain (LMCS), Long Parameter List (LPL), and Long Lambda Function (LLF). Each code smell was initially stored in its own dedicated comma-separated file. To streamline our analysis, we strategically merged all these code smells into a single comprehensive comma-separated file. This resulted in the creation of a dataset where each row can be associated with one or more class. This area of problem is known as multi-label classification [3]

During this merging process, a notable observation surfaced: the incidence of each code smell per project was relatively low, albeit substantial enough to warrant consideration as a significant issue. Recognizing the potential for dataset imbalance, we took proactive measures to address this concern. Specifically, to ensure a balanced representation, we carefully selected an equal number of rows representing both smelly and non-smelly instances for each code smell.

The resultant consolidated comma-separated file was aptly named the “code smell dataset”. This multi-labeled dataset underwent a rigorous and systematic workflow encompassing exploratory data analysis, data preprocessing, model training, and model evaluation.

Chapter 6

Methodology

6.1 A comprehensive comparative analysis between tools that are integrated with compiler and tools that uses machine learning models for refactoring

6.1.1 AI-Powered Tools for Refactoring

Machine learning and deep learning are becoming popular day by day and have been introduced to almost every aspect of computer science to make a developer's life easier. In recent years, artificial intelligence has slid into the software industry and has given rise to groundbreaking discoveries in automated detection of code smells and automated(semi) refactoring of such code smells. In this part of the paper we will look at some of the models and deep learning based tools that have been developed to address the issue. Firstly, Aniche M. et al [30] have created a dataset by taking projects from Apache, github and F-Droid to assess how machine learning models such as random forest, SVM, naive bayes, decision tree and neural network perform in providing recommendations for refactoring. After evaluating the models, it was observed that random forest outperformed all the other models with accuracy of 93%, 90% and 94% in providing opportunities for refactoring on the class, method and variable level respectively [30]. All other models have also performed well.

In addition to the existing machine learning models, many tools integrated with deep learning models have been built to address this problem. One such tool is Rmove [44], which has nine classifiers integrated in it including CNN, LSTM, and GRU. This tool learns both structural and semantic information from a given code snippet and suggests which portion of code requires the move method refactoring. Moreover, AntiCopyPaster [44], which is a plugin for IntelliJ IDEA, also uses CNN to detect duplicate chunks of code whenever a developer copy pastes a code or writes a duplicate code. The model immediately suggests extract method refactoring on those parts of the code. CNN's contribution does not stop here, this model is also seen in another tool called feTruth. feTruth determines the correct class a method should be moved to. Similar to feTruth, Graph neural network models [44] can be used to detect relationships between method and class and determine which method should be moved to which class. Neural network [44], even though related to graph

neural network models, takes a different approach in identifying in which class a method should be moved. The authors [44] suggested that if the neural network finds a method that is more interested in another class for more than two times the output of the neural network model would be 0.5. ANN has also been used as a fitness function [11] in a search-based learning algorithm as a newer approach to software refactoring. This approach has been evaluated using six open source systems and it was observed that this approach resulted in a precision of 80%. As a deep learning model is being used, manual intervention is minimized as the ANN can suggest which portions of the software needs refactoring [11]. Another study [34] shows that ANN can be integrated into existing refactoring tools to detect code smell and provide recommendations for refactoring. Before ANN was integrated into the refactoring tool of Eclipse IDE, the tool could not detect code smells such as duplicated code [34]. However, this problem was mitigated by ANN and the tool now provided a finer grain detection of code smells.

The deep learning techniques to suggest refactoring are enhanced with the use of state-of-the-art embedding techniques [44] such as CodeBERT, GraphCodeBERT, CodeGPT, CodeT5, PLBART, and CoTexT that are used to find appropriate lines of code that can be extracted to make a new method. All these embedders created an abstract syntax tree of the lines of code. Deep learning models such as CNN and LSTM were trained on the abstract syntax tree and were tested to see how they performed in recommending extract method refactoring techniques [44].

End-to-end code transformation is the process of actually taking the code and refactoring the code based on the refactoring suggestions while making sure the code behaviour does not change [44]. One such deep learning approach was taken to refactor a programming language called the Erlang. Their tool consists of two parts: localizer, detects code patterns that are same and can be used in different contexts, and Refactoring component, transforms the same code to only exist once. The source code is transformed to a sequence of tokens, which are the input of the neural network model. Then the neural network finds the non-idiomatic parts of the code. These non-idiomatic tokens are then used as the inputs to a recurrent seq-2-seq model with attention mechanism [44]. This recurrent model outputs the idiomatic code tokens which is the refactored version of the code. The localizer achieved an accuracy of 99% and the Refactoring component achieved an accuracy of 99.46% [44].

Compilers are not really good at suggesting variable names based on the context of the code. To tackle this challenge, BERT architecture [44] has been implemented in a tool called RefBERT to suggest variable names based on the surrounding codes. The RefBERT [44] model uses the masked language modeling analogy to refactor variable names. It blanks the variable name and looks at the surrounding codes and determines the variable name that is the most suitable in place of the blank. After evaluation RefBERT came up to be a reliable tool to use in suggesting context based variable names.

Another huge benefit of introducing deep learning based refactoring is that it can identify parts of a UML class diagram that can be refactored to reduce the number

of classes present in the system [40]. ANN is used in this case to have a precision of 0.87, recall of 0.93, and F1 score of 0.90 in identifying functional decomposition in an UML class diagram [40].

6.1.2 IDE Plug-in Tools For Refactoring

For many years, researchers and programmers have created and researched many praiseworthy tools and libraries that were integrated with the IDE that is prominent in refactoring smelly codes in a program. In this section we have thoroughly analyzed and studied previous works of the researchers that gave us valuable insights on how these refactoring tools worked, and how they have performed in mitigating code smells and providing a more cleaner software. At first, in the paper [6], researchers have studied and analyzed how WitchDoctor an IDE performs in refactoring programs. Its automatic ability allows it to detect programmers coding activity and enables it to refactor the code before the programmer refactor by his/her own. The system comprises many complex computation paradigms like textual differencing, delayed AST node mapping, declarative specification language, pattern-matching algorithms, and selective rollback. This complex system has made it possible for WitchDoctor for faster, reliable and robust refactoring suggestions where it is successful in simulating over 5000 refactoring operations across different projects. WitchDoctor has a commendable ability where it can tackle programs that are not parsed and incomplete. In addition, it is quick in adjusting to an abrupt change in the program's syntax which proves its accuracy.

Moving forward, the researchers in this paper [10] have introduced ClangMR which is greatly parallelized and semantically aware. ClangMR refactors C++ programs and it utilizes the broad semantic understanding from C++ abstract syntax tree to make the refactoring process well versed. In addition, the system also has the ability to parallelize across multiple computer systems at once through leveraging the MapReduce framework which allows ClangMR to refactor millions of lines of code at a quick rate. As a result ClangMR is successful in refactoring huge codebases without breaking the rules of semantics making it less prone for introduction of new errors in the program. Apart from its speed, the system is also advantageous in terms of scaling as it can transform 35000 function call sites where the size of the C++ program is around 100 millions.

In De Stefano et al.'s paper [32], they have developed a novel method in detecting and refactoring code smells that is that they have created a Java program plug-in named Casper which is integrated to IntelliJ IDEA. Casper can identify and refactor four kinds of code smells like Feature Envy, Misplaced Class, Blob and Promiscuous Package. Thus, the system's semi-automatic guidance does not require for programmers to refactor manually. In terms of accuracy, its accuracy is challenged in relation to detecting code smell as the tool does not ensure whether it correctly identifies genuine instances of code smells in source code or not. Also the refactoring process does not provide the most accurate result as it fails to ensure the suggested altered program effectively addresses the identified code smells without creating new errors. Even the researchers think that casper can be validated and improved through user feedback.

Furthermore, there have been many comparison analyses of how refactoring tools perform which are binded with integrated environment developments (IDEs). Regarding this, Martin Drozdz et al. 's [2] excellently portrayed how the refactoring tools in intelliJ's IDEA, IBM's Eclipse, and Sun's Netbeans performed. According to their research, IntelliJ IDEA 5.0 is praised for its superior ease of use, productivity, and comprehensive static code analysis tools, which include nearly 500 checks that allow for precise identification of code smells and refactoring requirements. Eclipse 3.1, while robust with a comparable range of refactorings, is described as slightly less user-friendly and productive than IDEA, with a lower emphasis on the accuracy of its refactoring tools. Netbeans 4.0 is criticized for its limited refactoring capabilities, which support only four refactorings and lack essential features such as the extract method, making it less effective in identifying and addressing code smells than IDEA and Eclipse. On the other hand, another brilliant work [14] has emerged connected to tool based automatic refactoring by IDEs. Francesca Arcelli et al. 's have done a comprehensive study on four refactoring tools like RefactorIT, JDeodorant, Eclipse and IntelliJ IDEA. In compliance with their experimentation, they have claimed that Eclipsed outperformed all other tools in refactoring with high accuracy and reliability. IntelliJ also performed well in refactoring but they observed that the tool occasionally dysfunctions and not that efficient in refactoring. RefactorIT performed well in removing code smells, but it lacked some features and had reliability issues due to crashes. JDeodorant struggled with accuracy and speed, introducing errors during refactoring and offering less reliable recommendations than Eclipse and IntelliJ IDEA.

Lastly, Tsantalis et al.'s [26] did an extensive study on how JDeodorant, a Java programming language based, performs in eliminating code smells that are in line with object-oriented programs. JDeodorant has been evolved over the years and it pivots on identifying which part of the program needs to be refactored, choosing which refactorings algorithm to use, ensuring behavior preservation, carrying out the refactorings of the identified smelly portion, evaluating the impact on quality characteristics, and ensuring consistency between the refactored code and other types of software fragments. While testing, JDeodorant reached a specific level of precision and recall for detecting specific code smells. For instance, while refactoring using the Move Method technique, JDeodorant achieved an average precision of 0.38 and an average recall of 0.25.

6.1.3 Comparative Analysis between IDE plugins /Compilers and Artificial Intelligence to Refactor Code Smells

In this section we have highlighted and emphasized on how machine learning models performed exceptionally compared to tools that are in compliance with IDE. We have done a complete investigation on a considerable amount of previous works relating to this matter and finally we have concluded that machine learning refactoring approaches outperforms tool based refactoring in three arguments. To begin with, some of the IDE-plugins do not provide any numeric evidence in terms of accuracy, precision and other performance metrics on how they have performed. In addition some of the tools have not been compared with other other tools to validate its

actual performance. For example- WitchDoctor [6] and ClangMR [10] being fast, reliable and robust in refactoring but there is no solid proof in the respective papers on how accurately these tools performed based on numbers or percentage metrics. On the other hand, research papers relating to artificial intelligence based refactoring are supported by performance metrics. For example, the figure 6.1 below shows how Aniche et al. [30] have given detailed experimental results which validates its accuracy.

	Logistic Regression			SVM (linear)			Naive Bayes (gaussian)			Decision tree			Random Forest			Neural Network		
	Pr	Re	Acc	Pr	Re	Acc	Pr	Re	Acc	Pr	Re	Acc	Pr	Re	Acc	Pr	Re	Acc
Class-level refactorings																		
Extract Class	0.78	0.91	0.82	0.77	0.95	0.83	0.55	0.93	0.59	0.82	0.89	0.85	0.85	0.93	0.89	0.80	0.94	0.85
Extract Interface	0.83	0.93	0.87	0.82	0.94	0.87	0.58	0.94	0.63	0.90	0.88	0.89	0.93	0.92	0.92	0.88	0.90	0.89
Extract Subclass	0.85	0.94	0.89	0.84	0.95	0.88	0.59	0.95	0.64	0.88	0.92	0.90	0.92	0.94	0.93	0.84	0.97	0.89
Extract Superclass	0.84	0.94	0.88	0.83	0.95	0.88	0.60	0.96	0.66	0.89	0.92	0.90	0.91	0.93	0.92	0.86	0.94	0.89
Move And Rename Class	0.89	0.93	0.91	0.88	0.95	0.91	0.69	0.94	0.76	0.92	0.95	0.94	0.95	0.95	0.95	0.88	0.94	0.91
Move Class	0.92	0.96	0.94	0.90	0.97	0.93	0.67	0.96	0.74	0.98	0.96	0.97	0.98	0.97	0.98	0.92	0.97	0.94
Rename Class	0.87	0.94	0.90	0.86	0.96	0.90	0.63	0.96	0.69	0.94	0.91	0.93	0.95	0.94	0.94	0.88	0.94	0.91
Method-level refactorings																		
Extract And Move Method	0.72	0.86	0.77	0.71	0.89	0.76	0.63	0.94	0.69	0.85	0.75	0.81	0.90	0.81	0.86	0.79	0.85	0.81
Extract Method	0.80	0.87	0.82	0.77	0.88	0.80	0.65	0.95	0.70	0.81	0.86	0.82	0.80	0.92	0.84	0.84	0.84	0.84
Inline Method	0.72	0.88	0.77	0.71	0.89	0.77	0.61	0.94	0.67	0.94	0.87	0.90	0.97	0.97	0.97	0.77	0.85	0.80
Move Method	0.72	0.87	0.76	0.71	0.89	0.76	0.63	0.93	0.70	0.98	0.87	0.93	0.99	0.98	0.99	0.76	0.84	0.78
Pull Up Method	0.78	0.90	0.82	0.77	0.91	0.82	0.68	0.95	0.75	0.96	0.88	0.92	0.99	0.94	0.96	0.82	0.87	0.84
Push Down Method	0.75	0.89	0.80	0.75	0.90	0.80	0.66	0.94	0.73	0.97	0.76	0.87	0.97	0.83	0.90	0.81	0.92	0.85
Rename Method	0.77	0.89	0.80	0.76	0.90	0.80	0.65	0.95	0.71	0.78	0.84	0.80	0.79	0.85	0.81	0.81	0.82	0.81
Variable-level refactorings																		
Extract Variable	0.80	0.83	0.82	0.80	0.83	0.82	0.62	0.94	0.68	0.82	0.83	0.82	0.90	0.83	0.87	0.84	0.89	0.86
Inline Variable	0.76	0.86	0.79	0.75	0.87	0.79	0.60	0.94	0.66	0.91	0.85	0.88	0.94	0.96	0.95	0.81	0.82	0.82
Parameterize Variable	0.75	0.85	0.79	0.74	0.86	0.78	0.59	0.94	0.65	0.88	0.81	0.85	0.93	0.92	0.92	0.80	0.83	0.81
Rename Parameter	0.79	0.88	0.83	0.80	0.88	0.83	0.65	0.95	0.71	0.99	0.92	0.95	0.99	0.99	0.99	0.82	0.87	0.84
Rename Variable	0.77	0.85	0.80	0.76	0.86	0.79	0.58	0.92	0.63	0.99	0.93	0.96	1.00	0.99	0.99	0.81	0.84	0.82
Replace Variable With Attribute	0.79	0.88	0.82	0.78	0.89	0.82	0.64	0.95	0.71	0.90	0.84	0.88	0.94	0.92	0.93	0.79	0.92	0.84

Figure 6.1: Evaluation of Machine Learning Models on Refactoring

Moving forward to our next argument, the machine and deep learning models are authenticated by comparing with IDE based tools. Some of the examples are:

- Rmove machine learning tool was compared with other compiler based tools such as PathMove, JDeodorant, Jmove it was observed that it had an increase in precision, recall, and F-measure from 14%-36%, 19%-45%, and 27%-44% while refactoring Feature Envy code smell [44].
- Another CNN based tool feTruth to detect feature envy and suggest the class to which a method should be moved. feTruth has beaten JDeodorant and JMove in refactoring where feTruth showed that it had an accuracy of 93.1% which is greater than JDeodorant's 80% and JMove's 87.5% [44].
- Graph NN provided an algorithm to determine in which class the method should be moved to and when compared with JDeodorant and JMove the accuracy improved by 5.13% and 11.0% [44].

In our last argument, we concluded that unlike machine learning, compilers use a rule based approach to refactor code smells [19]. Machine learning approaches are data-driven and identify code smells based on various software metrics. As the machine learning models are data-driven they can figure out the parts of the code that hinders the software quality the most. Moreover, models also help us give a more in-depth analysis of which code smells affect the software quality. This would help developers to handle the most severe type of code smells first. As machine

learning learns from various types of projects and source codes, in contrast to IDE plugins it does not rely on rule based refactoring [19]. Also, since tools that are compliant with the compiler use a rule based approach they cannot adapt to and handle changes in the codebase. To provide an example, tools like Netbeans 4.0, IntelliJ and Refactor IT [2], [14] cannot refactor certain code smells and collapses on its own. In lieu, there are no traces of machine learning models failing to refactor or crashes in any of the surveyed papers as machine or deep learning models are more generalized and can handle a variety of software systems. Therefore, for better understanding how machine learning models have a competitive edge over IDE plug-ins a Table 6.1 has been created.

Machine Learning Tool	Models Used	Comparison Scope	Comparison Criteria	Compared IDE Tools	Numeric Improvements
Rmove [44]	Decision Tree, Naive Bayes, SVM, Logistic Regression, Random Forest, Extreme Gradient Boost, CNN, GRU, LSTM	Feature Envy	Precision, Recall, F-measure	PathMove, JDeodorant, Jmove	14%-36%, 19%-45%, 27%-44%
AntiCopy-Paster [44]	CNN	Long Method	F - measure	N/A	0.82
feTruth [44]	CNN	Feature Envy	Accuracy	JDeodorant, JMove	13%, 5.6%
RefBERT [44]	BERT	Variable Naming	N/A	N/A	N/A
N/A	Graph Nueral Network [44]	Feature Envy	Accuracy	JDeodorant, JMove	5.13%, 11.0%
N/A	Neural Network [44]	Feature Envy	Accuracy	JDeodorant, JMove	Accuracy is 75% which is better than both
N/A	Embedding Technique with LSTM [44] & CNN	Long Method	N/A	GEMS, JExtract, SEMI, JDeodorant, Segmentation	Outperform All IDE tools
N/A	ANN as Fitness Function [11]	Code Smells	Precision	JDeodorant, JMove	15%, 6%

Table 6.1: Comparision of various ML Tools and models with IDE plugins

6.2 Code Smells and Metrics

Our research centers on the detection and reduce specific code smells, each resulting in unique challenges to software maintainability:

Large Class (LC): A class that contains excessive number of lines [16].

Long Method (LM): A method that is huge in length making the code difficult to maintain [16].

Long Parameter List (LPL): A method that takes in a lot of parameters as argument [16].

Long Message Chain (LMCS): Occurs when multiple methods are called using dot. This leads to lengthy calling of methods [16].

Long Lambda Function (LLF): Identifies lambda functions with an excessive number of characters, deviating from their intended purpose as concise inline functions [16].

For each code smell we selected, there is a specific metric or number of metrics that determine the presence of a code smell. The metrics are taken from previous literature [16]. Table 6.2 illustrates the code smell, at which level it occurs, the metric, and the threshold of the metric at which the code smell occurs.

Code Smell	Level	Criteria	Metric
Large Class (LC)	Class	$CLOC \geq 35$	CLOC: Class Line of Codes
Long Method (LM)	Function	$MLOC \geq 50$	MLOC: Method Line of Codes
Long Message Chain (LMCS)	Expression	$LMC \geq 4$	LMC: Length of Message Chain
Long Parameter List (LPL)	Function	$PAR \geq 5$	PAR: Number of Parameters
Long Lambda Function (LLF)	Expression	$NOC \geq 70$	NOC: Number of Character

Table 6.2: Metric-based Code Smells for Python

6.3 Artificial Neural Network (ANN)

In the development of our machine learning model for the multi-labeled dataset, a neural network emerged as the preferred choice. With the task of generating five outputs corresponding to code smell probabilities based on given inputs, we designed the architecture with key decisions already in place.

For the neural network architecture, the number of input and output nodes was predetermined. The Rectified Linear Unit (ReLU) activation function was chosen for the hidden layers, as it produces less computation due to some neurons being inactive [21], and the Sigmoid activation function for the output layer, as this is a classification problem sigmoid functions help in transforming the output to probability space thus performing better than other activation function [21]. However, determining the optimal number of hidden layers and nodes posed a challenge. Beginning with the mean of inputs and outputs as the number of nodes, we iteratively adjusted this count to find the most effective configuration.

Having finalized the architecture, we proceeded to test the model using different versions of our dataset. Initially, irrelevant columns were dropped, resulting in an accuracy range of 45% to 50%. Subsequently, scaling the dataset improved accuracy to around 60%

The optimal architecture, shown in figure 6.2, materialized with two hidden layers, the first comprising twelve nodes and the second six nodes. The dataset's refinement involved removing outliers, scaling the data, and achieving balance by duplicating rows. Upon evaluation, the model demonstrated an impressive accuracy of 87% to 90%. Further validation was conducted by testing the model with random inputs, producing anticipated results. This comprehensive approach to model development and testing underscores the effectiveness of the chosen neural network architecture and dataset refinement techniques.

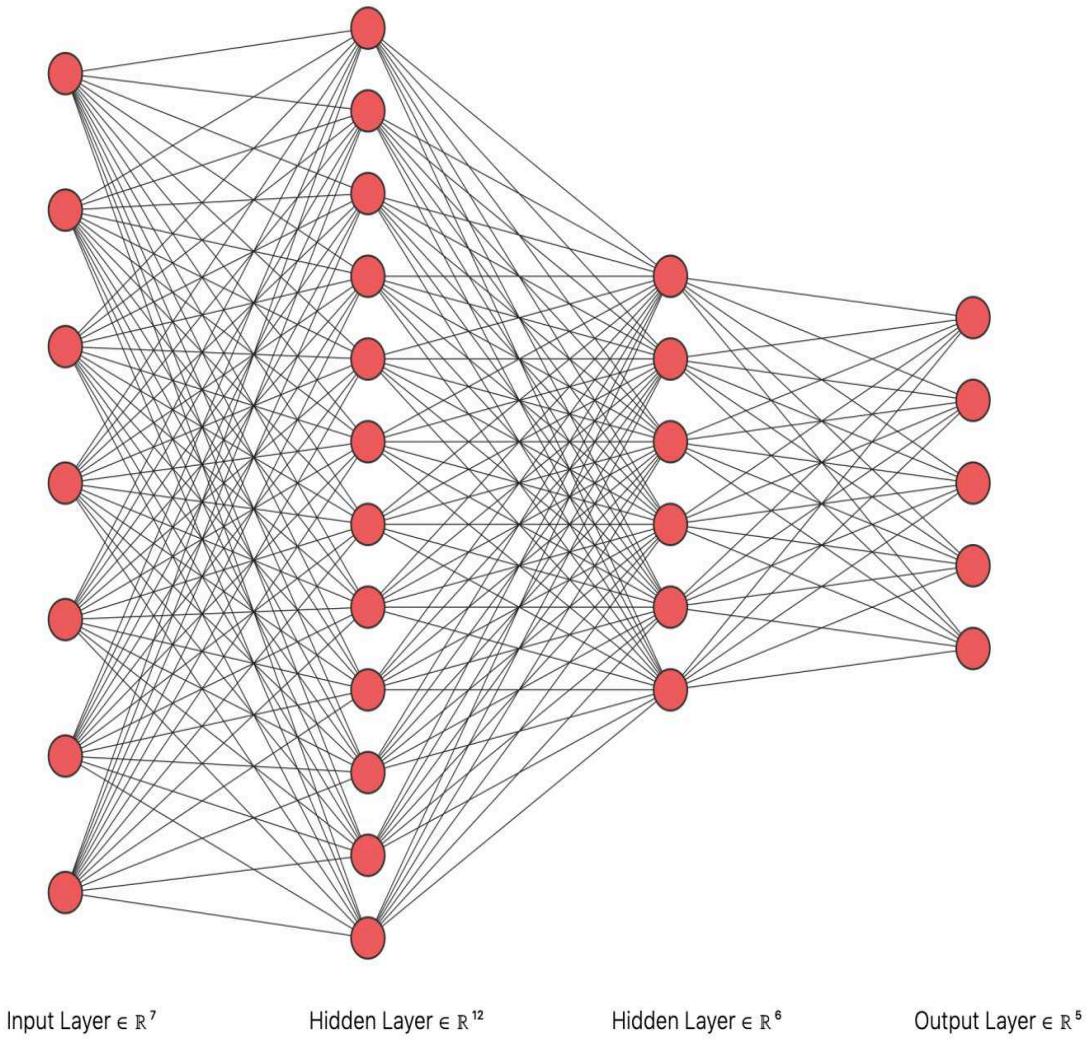


Figure 6.2: Architecture of The Neural Network

6.4 Ensemble Learning using Label Powerset

6.4.1 Label Powerset

In a multi-label classification problem, a single instance can belong to more than one or more classes. To deal with such multiple-label datasets, problem transformation or algorithm adaptation approaches can be considered. For our dataset, we have used a problem transformation method. The algorithm that is used in the problem transformation method converts the multi-label learning technique into one or more single-label classification techniques, where each transformed problem can be viewed as a typical binary classification task [12]. Further, many algorithms fall under the category of problem transformation methods like binary relevance, label powerset, and classifier chains. For our multi-label classification problem, we have chosen the label powerset algorithm. The label powerset algorithm converts the task of classifying multiple labels into classifying all the probable combinations of labels. This works by taking all the unique subgroups of multiple labels that are present in the training dataset and creating each subgroup a class attribute for the classification problem. Figure 6.3 below shows the classification procedure for the label powerset.

The diagram illustrates the label powerset transformation process. On the left, a multi-label dataset is shown in a table with columns X, Y₁, Y₂, Y₃, and Y₄. The rows represent instances X₁ through X₆. The values in the Y columns indicate which labels are present for each instance. An arrow points from this table to the right, indicating the transformation. On the right, a single-label dataset is shown in a table with columns X and Y. The same six instances are listed, but now each instance has a single label value in the Y column. The labels assigned are 1 for X₁, 2 for X₂, 3 for X₃, 1 for X₄, 4 for X₅, and 2 for X₆.

X	Y ₁	Y ₂	Y ₃	Y ₄
X ₁	0	1	0	0
X ₂	0	1	1	0
X ₃	1	0	0	0
X ₄	0	1	0	0
X ₅	1	1	1	1
X ₆	0	1	1	0

→

X	Y
X ₁	1
X ₂	2
X ₃	3
X ₄	1
X ₅	4
X ₆	2

Figure 6.3: Label Powerset example

6.4.2 Ensemble Learning Algorithms

After converting the multi-label classification problem using the label powerset, we can use traditional ensemble learning techniques. Concerning our problem, we have used three ensemble models, which are:

- AdaBoost: AdaBoost, which is short for adaptive boosting, is one of the ensemble techniques that accumulates the output of weak learners to create a strong learner. It is known that for classifying binary kinds of problems, AdaBoost won the race. In addition, it is the most familiar boosting approach for such two-fold classification problems [37].

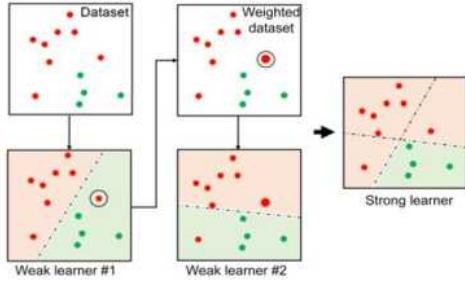


Figure 6.4: AdaBoost example

- XgBoost: XgBoost, which is short for extreme gradient boosting, is a popular ensemble technique that was developed by Tianqi Chen [37]. XgBoost uses decision trees as its base learners and its goal is to minimize the objective function. It is well known for optimizing the gradient-boosting algorithm.

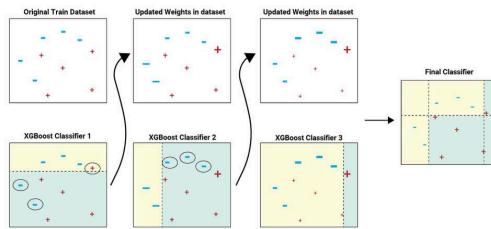


Figure 6.5: XgBoost example

- Random Forest: This is one of the most widely used algorithms for classification and regression problems. It is also an ensemble model that joins multiple decision trees in order to enhance accuracy and lower overfitting [41].

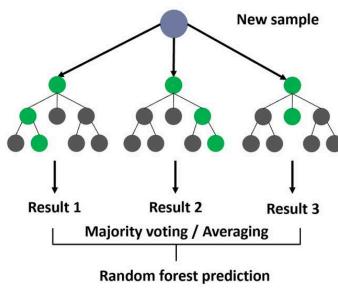


Figure 6.6: Random Forest example

6.4.3 Model Implementation and their Performance

Regarding our multi-label code smell classification problem, we first imported required libraries like Label Powerset, AdaBoost, XgBoost, and Random Forest from the Python scikit-learn library. Moving on, the input features like the code smell metrics were labeled as the x-variable and the code smells were labeled as the output y-variable. Then the tertiary dataset is divided for training and testing, with 80%

of the dataset for training and the remaining for testing. To run the algorithms we have to first choose our base classifier like AdaBoost, XgBoost, and Random Forest. In the process of the implementation, the Label powerset function then creates all the subset of combinations like {Large Class}, {Large Class, Long Method}, {Long Method, Long Parameter List, Long Lambda Function} etc, where all the combinations become a distinct category and the base classifiers try to predict them all at once. In the next phase, we used several performance metrics to understand how better the base classifiers were in predicting the code smells from the Python files. We have utilized accuracy, precision, F1, recall, and hamming loss as evaluation metrics. The performance of the ensemble techniques is outlined below in the table:

Ensemble Models	Accuracy	Precision	F1-Score	Recall	Hamming Loss
AdaBoost	69%	96%	83%	73%	0.06
XgBoost	100%	100%	100%	100%	0.00000798
Random Forest	100%	100%	100%	100%	0.00000533

Table 6.3: Performance metrics for Ensemble Learning Models

From Table 6.3, we can evaluate that XgBoost and Random Forest were absolute best in detecting code smells with 100% accuracy, precision, F1-score and Recall. However, AdaBoost could not perform well in identifying the code smells. In addition, XgBoost and Random Forest's hamming loss value was also very low compared to AdaBoost.

6.5 Refactoring

6.5.1 Refactoring of Long Method

As RQ1 and RQ2 which have already been addressed, RQ3 addresses the fact of incorporating an intelligent algorithm to automate the process of refactoring code smell. According to our analysis, Large Class and Long Method were the most popular contributing bad smells in python files. Hence, we have chosen Long Method as our prime code smell to restructure it. Almost every author in their papers has mentioned Martin Fowler and Kent Beck [20], [28], [38], [39], [42] who brought enlightenment in the software code smells and refactoring process. Martin and Kent [25] vividly describes the art of refactoring. It is a process of creating a change in a software structure without changing the external composition of the code, however bringing an enhancement in the internal composition of code. Refactoring aids in eradicating code smells, improving the software quality of the system and diminishes the cost of maintaining a code base. Therefore, reducing the possibility of introducing new software bugs and also identifying bugs if there are any. It is important to understand that refactoring is not synonymous to rewriting a code script as refactoring fails to alter the functionality of the code script [28]. Refactoring a python Long Method code smell manually can have several drawbacks like immense time consumption, tedious and prone to making errors while refactoring. As a result an automated process of refactoring will bring ease in developers' minds.

6.5.2 Long Method and its refactoring solution

Long method is one of the most rampant code smells in almost every software project [39]. As the name suggests, it deals with methods (functions) in programs and it falls under the category of ‘bloaters’. Bloaters are referred to code, methods and classes that its length have increased to such an enormous extent that it becomes extremely complex to work with. Long method gives developers a hard time in reading and understanding the code scripts as the method consists of multiple conditions, loops, variable declarations and data operations. The problem of long methods can be decoded by decreasing the complexity of the method. This is done through pointing out those parts of the function that need explanation and separate it into a new method [28].

Several techniques are established by Martin and Kent [25] to tackle long methods and (Agnihotri & Chug, 2020) have spotted the most familiar refactoring approaches like move method, extract class and extract method. In our study, we have used the principles of the extract method in order to build the architecture of our automated refactoring tool. Extract method is the process where part of the method can be divided and a new method is created for that part. Codes that need explanation are extracted and pasted in the new method. In addition, the necessary aspects like variables and parameters of the extracted method are exported as new parameters to the newly built method. (Fowler & Kent, 2018) have explained the benefits of extracting a method. Those benefits are that the extract method gives rise to the probability of other methods to use a method where it is properly refactored and improves the readability of the higher-level methods.

6.5.3 Detailed Architecture of the Proposed Refactoring Algorithm

In this segment, we will be presenting and highlighting every detail of our proposed model that aims to mechanize the entire procedure of refactoring a Long Method. Taking conceptual ideas of extracting a method from (Fowler & Kent, 2018) book and the following related research papers [1], [4] on refactoring Long Methods of ‘Java’ programs. We have formulated our proposed approach on refactoring python Long methods.

The flow chart presented in Figure 6.7 is the high-level design of our suggested algorithm. In the first step, a python file containing long methods is given as an input to our main function which then carries out four important mechanisms. In the second step, the structure of the input file is checked like the indentation and spacing of the file. If there is any indentation error then it is corrected before moving to the next course of action. Moving on to the third and fourth step, long methods are thoroughly examined and conditional statements are found out and removed from the long method and placed in the new method. Lastly, the input file is refactored where complex if-else statements are converted into new methods (basic statements) and are called in the input file. Thus, the system reduces the overall line numbers of the input long method. The following sections give a comprehensive explanation of each step of the suggested algorithm.

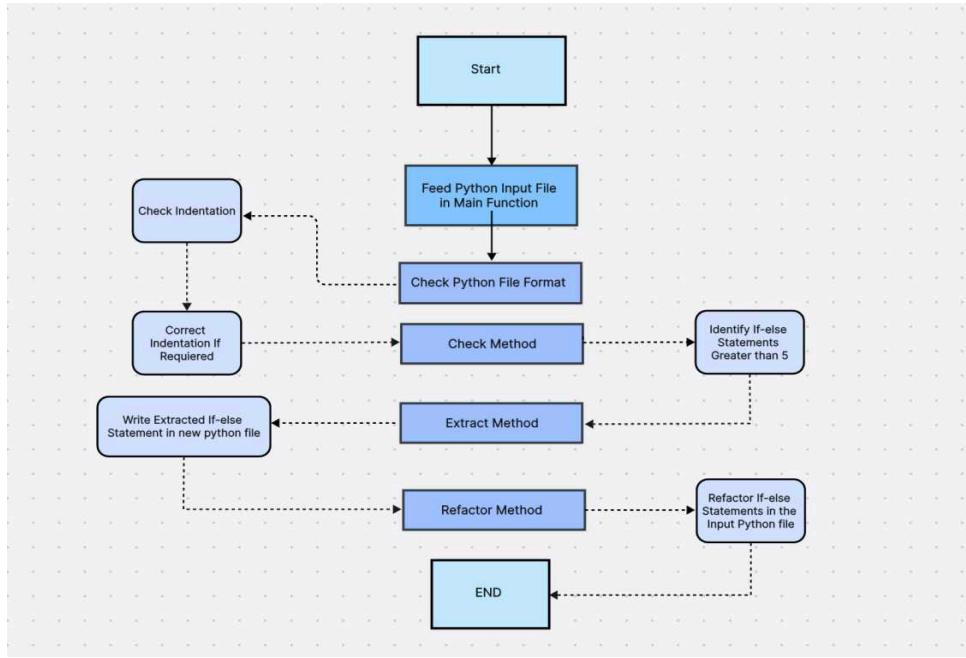


Figure 6.7: Flow chart of the naive refactoring algorithm

A. Exemplifying the Input File

To illustrate the mechanism of the algorithm we have created a simple program that consists of a long method of our own. This python long method is used to classify a person based on age, gender and income. The size of the method is 67 lines which exceeds the threshold of MLOC as discussed in Table 5.1. For better apprehension and automation of the refactoring process we have only considered conditional (*if-else*) statements that need to be extracted. As a result, the newly created method contains a gargantuan if-else statement making the method complex to refactor. The following function shown in listing 6.1 will be used as an example source code to elucidate the framework of the algorithm and it is referred as `example_code.py`.

```

1 def classify_person(age, gender, income):
2     for i in range(3):
3         if age < 18:
4             if gender == 'male':
5                 if income < 20000:
6                     return 'Young male with low income'
7                 elif 20000 <= income < 40000:
8                     num = 5
9                     return 'Young male with moderate income'
10                else:
11                    return 'Young male with high income'
12            else: # Gender is female
13                if income < 20000:
14                    # x += 1
15                    # y += 2
16                    return 'Young female with low income'

```

```

17     elif 20000 <= income < 40000:
18         return 'Young female with moderate income'
19     else:
20         return 'Young female with high income'
21     elif 18 <= age < 65:
22         if gender == 'male':
23             if income < 30000:
24                 return 'Middle-aged male with low income'
25             elif 30000 <= income < 60000:
26                 return 'Middle-aged male with moderate
27                     income'
28             else:
29                 return 'Middle-aged male with high income'
30     else: # Gender is female
31         if income < 30000:
32             return 'Middle-aged female with low
33                     income'
34         elif 30000 <= income < 60000:
35             return 'Middle-aged female with moderate
36                     income'
37         else:
38             return 'Middle-aged female with high
39                     income'
40     else: # Age is 65 or above
41         if gender == 'male':
42             if income < 25000:
43                 return 'Senior male with low income'
44             elif 25000 <= income < 50000:
45                 return 'Senior male with moderate income'
46             else:
47                 return 'Senior male with high income'
48     else: # Gender is female
49         if income < 25000:
50             return 'Senior female with low income'
51         elif 25000 <= income < 50000:
52             return 'Senior female with moderate
53                     income'
54         else:
55             return 'Senior female with high income'
56
57 # x = 5
58 if age < 18:
59     if gender == 'male':
60         if income < 20000:
61             return 'Young male with low income'
62         elif 20000 <= income < 40000:
63             # x = 5
64             return 'Young male with moderate income'
65         else:

```

```

60         return 'Young male with high income'
61 else: # Gender is female
62     if income < 20000:
63         return 'Young female with low income'
64     elif 20000 <= income < 40000:
65         return 'Young female with moderate income'
66     else:
67         return 'Young female with high income'

```

Listing 6.1: example_code.py

B. Composition of the Main File

This classify_person function is fed as an input to our main program. The main function then carries out and orchestrates sequentially the four key services in order to refactor the example long function classify_person. The four important branches of the main function include check_file_format.py, check_method.py, extract_method.py & refactor.py. However, if there is no large conditional statement in the input method then the main function will generate an output telling that there is no presence of conditional statement for extraction and refactoring. Figure 6.8 and the code snippet in listing 6.2 displays the functionality of the main function:

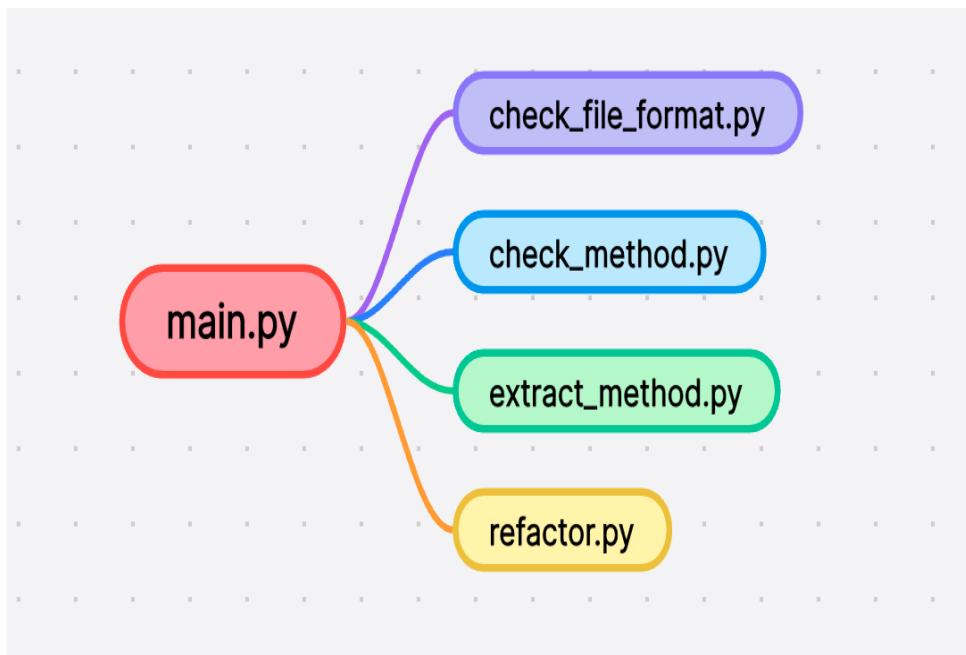


Figure 6.8: Structure of the main.py

```

1 from check_method import check_method
2 from extract_method import extract_method
3 from refactor import refactor
4 from check_file_format import check_file_format
5
6 def main(filename):

```

```

7 # Your main function to orchestrate everything
8 check_file_format(filename)
9 data, long_condition = check_method(filename)
10 print(long_condition)
11 if len(long_condition) != 0:
12     functions = extract_method(data, long_condition)
13     refactor(functions, long_condition, filename)
14 else:
15     print("There is no if statement that is between 6
16         and 50")
17
18 if __name__ == "__main__":
19     main('Example Codes/example_code.py')

```

Listing 6.2: main.py

C. Inspecting the input python file

In this section, the example_code.py is investigated exhaustively to find any unevenly indented code. This is done in a series of steps. At first the example_code.py is read line by line and we have used ‘rb’ (read binary) in order to not exclude any white spaces in the input file. Secondly, since the file is formatted to binary we have used ‘utf-8’ decoder to decode each binary line of code back into string data type. Moving on, each line is then added in an array data structure and the array is passed in a loop for checking and fixing indentation error. Lastly, the modified strings of lines are appended in another array called new_lines and it is further written in the existing input python file. For instance, there has been an indentation error in the line 4 and 61 of the example_code.py which will be formatted precisely after the implementation of the file formatting tool. The following listing 6.3 and 6.4 represent how check_file_format is capable to format the input python file:

```

1 def classify_person(age, gender, income):
2     for i in range(3):
3         if age < 18:
4             if gender == 'male':
5                 if income < 20000:
6                     return 'Young male with low income'
7                 elif 20000 <= income < 40000:
8                     num = 5
9                     return 'Young male with moderate income'
10                else:
11                    return 'Young male with high income'
12            else: # Gender is female
13                if income < 20000:
14                    # x += 1
15                    # y += 2
16                    return 'Young female with low income'

```

```

17     elif 20000 <= income < 40000:
18         return 'Young female with moderate income'
19     else:
20         return 'Young female with high income'
21     elif 18 <= age < 65:
22         if gender == 'male':
23             if income < 30000:
24                 return 'Middle-aged male with low income'
25             elif 30000 <= income < 60000:
26                 return 'Middle-aged male with moderate
27                     income'
28             else:
29                 return 'Middle-aged male with high income'
30     else: # Gender is female
31         if income < 30000:
32             return 'Middle-aged female with low
33                     income'
34         elif 30000 <= income < 60000:
35             return 'Middle-aged female with moderate
36                     income'
37         else:
38             return 'Middle-aged female with high
39                     income'
40     else: # Age is 65 or above
41         if gender == 'male':
42             if income < 25000:
43                 return 'Senior male with low income'
44             elif 25000 <= income < 50000:
45                 return 'Senior male with moderate income'
46             else:
47                 return 'Senior male with high income'
48     else: # Gender is female
49         if income < 25000:
50             return 'Senior female with low income'
51         elif 25000 <= income < 50000:
52             return 'Senior female with moderate
53                     income'
54         else:
55             return 'Senior female with high income'
56
57 # x = 5
58
59 if age < 18:
60     if gender == 'male':
61         if income < 20000:
62             return 'Young male with low income'
63         elif 20000 <= income < 40000:
64             # x = 5
65             return 'Young male with moderate income'
66         else:

```

```

60             return 'Young male with high income'
61 else: # Gender is female
62     if income < 20000:
63         return 'Young female with low income'
64     elif 20000 <= income < 40000:
65         return 'Young female with moderate income'
66     else:
67         return 'Young female with high income'

```

Listing 6.3: example_code.py before applying check_file_format.py

```

1 def classify_person(age, gender, income):
2     for i in range(3):
3         if age < 18:
4             if gender == 'male':
5                 if income < 20000:
6                     return 'Young male with low income'
7                 elif 20000 <= income < 40000:
8                     num = 5
9                     return 'Young male with moderate income'
10                else:
11                    return 'Young male with high income'
12            else: # Gender is female
13                if income < 20000:
14                    # x += 1
15                    # y += 2
16                    return 'Young female with low income'
17                elif 20000 <= income < 40000:
18                    return 'Young female with moderate income'
19                else:
20                    return 'Young female with high income'
21        elif 18 <= age < 65:
22            if gender == 'male':
23                if income < 30000:
24                    return 'Middle-aged male with low income'
25                elif 30000 <= income < 60000:
26                    return 'Middle-aged male with moderate
27                        income'
28                else:
29                    return 'Middle-aged male with high income'
30            else: # Gender is female
31                if income < 30000:
32                    return 'Middle-aged female with low
33                        income'
34                elif 30000 <= income < 60000:
35                    return 'Middle-aged female with moderate
                        income'
36                else:
37                    return 'Middle-aged female with high

```

```

            income'
36     else: # Age is 65 or above
37         if gender == 'male':
38             if income < 25000:
39                 return 'Senior male with low income'
40             elif 25000 <= income < 50000:
41                 return 'Senior male with moderate income'
42             else:
43                 return 'Senior male with high income'
44         else: # Gender is female
45             if income < 25000:
46                 return 'Senior female with low income'
47             elif 25000 <= income < 50000:
48                 return 'Senior female with moderate
49                     income'
50             else:
51                 return 'Senior female with high income'
51 # x = 5
52 if age < 18:
53     if gender == 'male':
54         if income < 20000:
55             return 'Young male with low income'
56         elif 20000 <= income < 40000:
57             # x = 5
58             return 'Young male with moderate income'
59         else:
60             return 'Young male with high income'
61     else: # Gender is female
62         if income < 20000:
63             return 'Young female with low income'
64         elif 20000 <= income < 40000:
65             return 'Young female with moderate income'
66         else:
67             return 'Young female with high income'

```

Listing 6.4: example_code.py after applying check_file_format.py

D. Analyzing the Long Method

The actions in the check method are segmented into three parts. The first and foremost task of the check method is to identify and keep count of the number of if-else statements. The criteria for being a long conditional statement is that if the size (*number of lines*) of the conditional statement is greater than 5, then the check method will regard it as a long conditional statement. All the local and global variables that are associated with the conditional statements need to be pointed out and passed as a parameter for the new function created. Additionally, the starting and ending line number of the if-else statement need to be accounted for so that the new function will be aware from which line number to call the new function. To keep track of the number of if-else statements

and variables, they are placed in a dictionary named long_condition. The key of the dictionary is the number of conditional statements while the values are the start and end of the conditional statement and variables. The figure 6.9 distinctly depicts the output of the long_condition from example_code.py. According to the example_code.py, there are two long if-else statements that are greater than 5, as a result there are two items in the dictionary. The first conditional structure had 4 parameters while the second condition had 3 parameters. Another important dictionary is also created in the check_method which is quite crucial in the extract_method section. This dictionary consists of the line number as key of the python dictionary and code on that particular line number as value for the python dictionary. Both of these dictionaries are passed as parameters of the extract method.

```
{1: [[3, 50], ['age', 'gender', 'income', 'num']], 2: [[52, 67], ['age', 'gender', 'income']]}
```

Figure 6.9: Output of the dictionary long_condition for the input exmaple

E. Configuration of the extract method

In the extract_method.py, a new method is automatically generated and placed in a new python file called new_method.py. For each of the long conditional statements, new functions are produced. This is done through recognizing the number of keys in the long_condition dictionary. Each of these new functions only contains the withdrawn if-else statements from the example_code.py and parameters of each of these functions are also taken out from the values of the long_condition. Although, for smooth and seamless building of new methods, the data dictionary is looped from the starting point of the if-else statement to the end point (*start and end points are passed as values in the long_condition*) so that any extra line of code apart from the conditional statements are omitted from the new functions that are created. The listing 5.5 clearly shows that the extract method is successful in bringing out two conditional statements and creating separate method for each of the conditions from the example_code.py. Listing 6.5 is the output for the extract function:

```

1 def method_1(age, gender, income, num):
2     if age < 18:
3         if gender == 'male':
4             if income < 20000:
5                 return 'Young male with low income'
6             elif 20000 <= income < 40000:
7                 num = 5
8                 return 'Young male with moderate income'
9             else:
10                return 'Young male with high income'
11        else: # Gender is female
12            if income < 20000:
13                # x += 1

```

```

14             # y += 2
15             return 'Young female with low income'
16         elif 20000 <= income < 40000:
17             return 'Young female with moderate income'
18         else:
19             return 'Young female with high income'
20     elif 18 <= age < 65:
21         if gender == 'male':
22             if income < 30000:
23                 return 'Middle-aged male with low income'
24             elif 30000 <= income < 60000:
25                 return 'Middle-aged male with moderate
26                     income'
27             else:
28                 return 'Middle-aged male with high income'
29         else: # Gender is female
30             if income < 30000:
31                 return 'Middle-aged female with low
32                     income'
33             elif 30000 <= income < 60000:
34                 return 'Middle-aged female with moderate
35                     income'
36             else:
37                 return 'Middle-aged female with high
38                     income'
39     else: # Age is 65 or above
40         if gender == 'male':
41             if income < 25000:
42                 return 'Senior male with low income'
43             elif 25000 <= income < 50000:
44                 return 'Senior male with moderate income'
45             else:
46                 return 'Senior male with high income'
47         else: # Gender is female
48             if income < 25000:
49                 return 'Senior female with low income'
50             elif 25000 <= income < 50000:
51                 return 'Senior female with moderate
52                     income'
53             else:
54                 return 'Senior female with high income'
55
56 def method_2(age, gender, income):
57     if age < 18:
58         if gender == 'male':
59             if income < 20000:
60                 return 'Young male with low income'
61             elif 20000 <= income < 40000:

```

```

57         # x = 5
58         return 'Young male with moderate income'
59     else:
60         return 'Young male with high income'
61     else: # Gender is female
62         if income < 20000:
63             return 'Young female with low income'
64         elif 20000 <= income < 40000:
65             return 'Young female with moderate income'
66         else:
67             return 'Young female with high income'

```

Listing 6.5: Output of extract method (*new_method.py*)

F. Refactoring the input long_method

As parameters the refactor function takes three parameters which are long_condition dictionary, the newly built extracted functions and the input example_code.py file. The example_code.py file is read from top to bottom and the refactor function taking help from the long_condition dictionary it notices the line number from where the new functions need to be placed. Eventually, on that particular line number the refactor function without making any mistake it calls the newly constructed functions.

```

1 from new_method import *
2 def classify_person(age, gender, income):
3     for i in range(3):
4         method_1(age, gender, income, num)
5     # x = 5
6     method_2(age, gender, income)

```

Listing 6.6: example_code.py after being refactored

From the above listing 6.6 it can be observed that our unsophisticated algorithm is successful in shortening the Long Method (*classify_person*) of the input file example_code.py. Since two conditions were present in the input file, two methods have been created, imported and called in the appropriate location and other lines of code have remained untouched. The number of lines has been drastically reduced from 67 to 6 lines which is significantly below the benchmark set for Long Method ($MLOC \geq 50$).

Chapter 7

Result Analysis

7.1 Analysis of Code Smell Detection

7.1.1 Data Analysis

To understand the relationship between the input features and the output class label, we have analyzed the data. This inspection of data was graphically inspected with the aid of Python libraries like ‘seaborn’ and ‘matplotlib’. At first, we plotted the number of code smells for each type of code smell in a bar chart to estimate the number of code smells that were present in each py file of the projects. In the bar charts represented in figure 7.1, the x-axis represents the type of class and the y-axis represents the amount of code smell in each file.

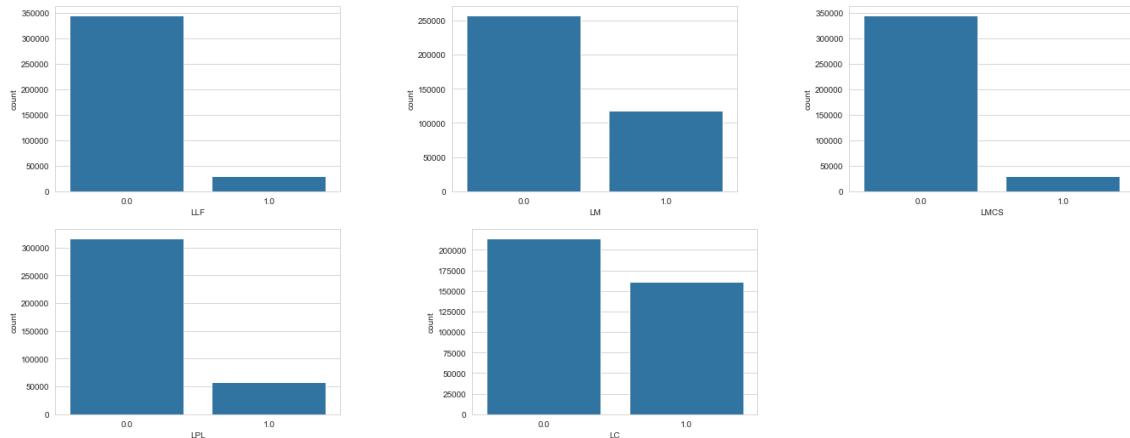


Figure 7.1: Class distribution of LLF, LM, LMCS, LPL & LC

After evaluating the bar charts, we can see that two bar graphs were plotted, where each bar graph shows the number of smelly (1) and non-smelly (0) py files. From the above graphs, we get a clear idea that most .py files contained Large Class (LC) and Long Method (LM) code smells. In contrast, very few py files contained code smells that belonged to the categories of Long Message Chain (LMCS), Long Parameter List (LPL), and Long Lambda Function (LLF).

7.1.2 Relationship between input features and the class label

To examine any relation between all the code smell metrics (input features) and the final class label, like the presence of Python code smell, we have plotted a scatter plot diagram. The scatter plot diagram gave us insightful information about their relationship. This was done through the use of the pairplot function of the seaborn library. The scatter plot diagrams for each code smell are shown in figure 7.2, 7.3, 7.4, 7.5 and 7.6.

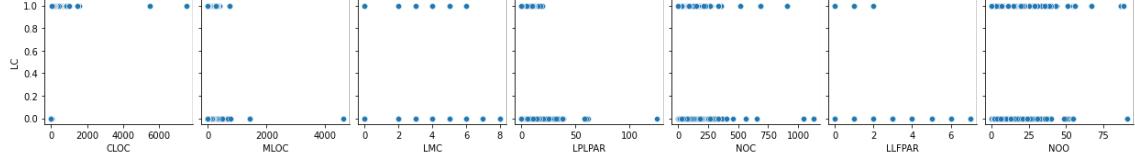


Figure 7.2: Relation between Large Class and other Code Smells

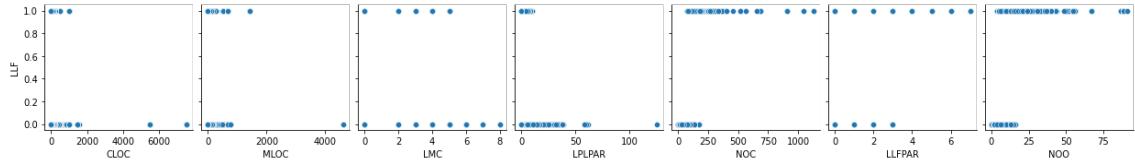


Figure 7.3: Relation between Long Lambda Function and other Code Smells

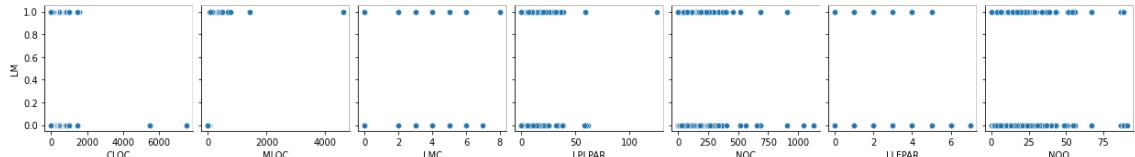


Figure 7.4: Relation between Long Method and other Code Smells

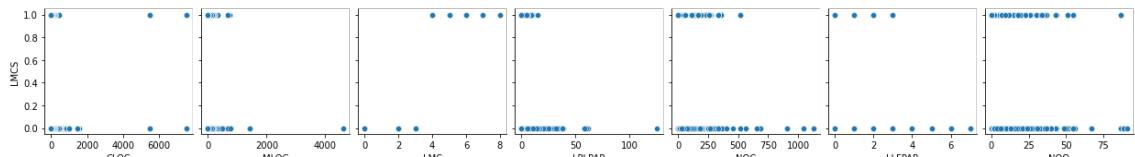


Figure 7.5: Relation between Long Message Chain and other Code Smells

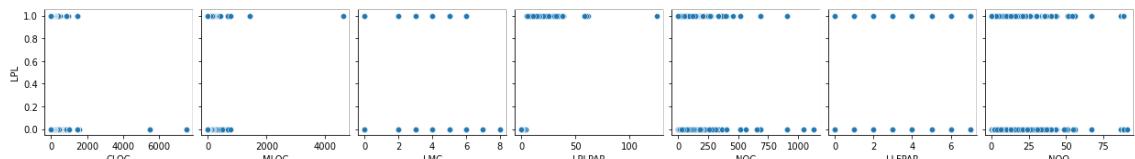


Figure 7.6: Relation between Long Parameter List and other Code Smells

From figures 6.2 to 6.6, we can understand that a code smell can occur not only based on its own particular metric but also due to the presence of other code smell metrics. For instance, in figure 6.5 above, we can see that most of the Large Class (LC) has occurred due to its own metric CLOC, but it is also evident that there are some of the Large Class smells that happened due to other metrics like MLOC, LMC, as well as the rest of the metrics.

7.1.3 Correlation of the code smells and their metrics

A statistical measure called correlation indicates how much two variables change together. A common way to represent correlation is with the Pearson correlation coefficient, which has a range of -1 to +1. A positive correlation means that if the value of one variable increases, then the value of the other variable also increases (value > 0). A negative correlation means that if the value of one variable increases, then the value of the other variable decreases (value < 0). If there is no linear relationship between the variables, then there is a correlation of 0 (value = 0). Therefore, we have created a heatmap to find out the correlation between the input features and class labels using the seaborn library. The heatmap is shown in figure 6.7.



Figure 7.7: Correlation Heatmap

From figure 7.7, we can see that there is a strong positive correlation between the code smell and their code smell metrics. Besides, there is a weak correlation between code smells and the code smell metrics that are not of their type.

7.2 Analysis of the Refactoring Algorithm

After designing our algorithm and building it using python, we ran the tool on four different python files with each file being more complex than its previous ones. These python files were drawn out from our primary dataset and each of these files consisted of a long method greater than the threshold ($MLOC \geq 50$). Even though we took a very naive approach of extracting long if-else statements to address the long method code smell, we saw some promising results. The results are discussed in the upcoming sub sections with codes from each file before and after going through our algorithm.

7.2.1 File 1: Basic Example with a long if-else statement

The example_code.py consisted of two simple and long if-else statements. The first statement was inside a for-loop and the second statement was on its own. The for-loop was added to check if our algorithm could handle different levels of indentations and some comments were also added to check if our algorithm could ignore these comments. After running our algorithm on the file shown in figure 7.8, it was seen that it could refactor it efficiently and elegantly.

```
1 v def classify_person(age, gender, income):
2 v     for i in range(3):
3 v         if age < 18:
4 v             if gender == 'male':
5 v                 if income < 20000:
6 v                     return 'Young male with low income'
7 v                 elif 20000 <= income < 40000:
8 v                     num = 5
9 v                     return 'Young male with moderate income'
10 v                else:
11 v                    return 'Young male with high income'
12 v            else: # Gender is female
13 v                if income < 20000:
14 v                    # x += 1
15 v                    # y += 2
16 v                    return 'Young female with low income'
17 v                elif 20000 <= income < 40000:
18 v                    return 'Young female with moderate income'
19 v                else:
20 v                    return 'Young female with high income'
21 v        elif 18 <= age < 65:
22 v            if gender == 'male':
23 v                if income < 30000:
24 v                    return 'Middle-aged male with low income'
25 v                elif 30000 <= income < 60000:
26 v                    return 'Middle-aged male with moderate income'
27 v                else:
28 v                    return 'Middle-aged male with high income'
29 v            else: # Gender is female
30 v                if income < 30000:
31 v                    return 'Middle-aged female with low income'
32 v                elif 30000 <= income < 60000:
33 v                    return 'Middle-aged female with moderate income'
34 v                else:
35 v                    return 'Middle-aged female with high income'
36 v        else: # Age is 65 or above
37 v            if gender == 'male':
38 v                if income < 25000:
39 v                    return 'Senior male with low income'
40 v                elif 25000 <= income < 50000:
41 v                    return 'Senior male with moderate income'
42 v                else:
43 v                    return 'Senior male with high income'
44 v            else: # Gender is female
45 v                if income < 25000:
46 v                    return 'Senior female with low income'
47 v                elif 25000 <= income < 50000:
48 v                    return 'Senior female with moderate income'

49         else:
50             return 'Senior female with high income'
51         # x = 5
52         if age < 18:
53             if gender == 'male':
54                 if income < 20000:
55                     return 'Young male with low income'
56                 elif 20000 <= income < 40000:
57                     # x = 5
58                     return 'Young male with moderate income'
59                 else:
60                     return 'Young male with high income'
61             else: # Gender is female
62                 if income < 20000:
63                     return 'Young female with low income'
64                 elif 20000 <= income < 40000:
65                     return 'Young female with moderate income'
66                 else:
67                     return 'Young female with high income'| You, 4
```

Figure 7.8: File 1 - Before Refactoring

```

1 | from new_method import *
2 | def classify_person(age, gender, income):
3 |     for i in range(3):
4 |         method_1(age, gender, income, num)
5 |         # x = 5
6 |         method_2(age, gender, income)

```

Figure 7.9: File 1 - After Refactoring

Figure 7.9 shows the file after refactoring, from this we observed that even though “num” is a local variable, which was declared inside the if-else statement, it got passed as a parameter to method_1. Moreover, the new methods that were created return a value thus it should have been stored in a variable rather than just calling it. Hence, some manual adjustment is needed to make the code work properly. Even though our algorithm did not work, it correctly identified the part of code that needed refactoring and almost provided a neat solution.

7.2.2 File 2: Example that does not require refactoring

We chose our second file, which did not require any refactoring. This was deliberately done to analyse how our tool tackled a code base that has small if-else statements.

```

1 | def __init__(self):
2 |     if hasattr(self, "_tracker"):
3 |         return
4 |
5 |     trainable_variables = []
6 |     non_trainable_variables = []
7 |     layers = []
8 |     metrics = []
9 |     seed_generators = []
10 |    self._tracker = tracking.Tracker(
11 |        {
12 |            "trainable_variables": [
13 |                lambda x: isinstance(x, backend.Variable) and x.trainable,
14 |                trainable_variables,
15 |            ],
16 |            "non_trainable_variables": [
17 |                lambda x: isinstance(x, backend.Variable)
18 |                and not x.trainable,
19 |                non_trainable_variables,
20 |            ],
21 |            "metrics": (lambda x: isinstance(x, Metric), metrics),
22 |            "layers": [
23 |                lambda x: isinstance(x, Layer)
24 |                and not isinstance(x, Metric),
25 |                layers,
26 |            ],
27 |            "seed_generators": [
28 |                lambda x: isinstance(x, backend.random.SeedGenerator),
29 |                seed_generators,
30 |            ],
31 |        }
32 |    )
33 |
34 |    if backend.backend() == "tensorflow":
35 |        # Remove attribute tracking for lists (TF-specific attribute)
36 |        _self setattr_tracking = getattr(
37 |            self, "_self setattr_tracking", True
38 |        )
39 |
40 |        self._trainable_variables = trainable_variables
41 |        self._non_trainable_variables = non_trainable_variables
42 |        self._layers = layers
43 |        self._metrics = metrics
44 |        self._seed_generators = seed_generators
45 |
46 |    if backend.backend() == "tensorflow":
47 |        # Reset attribute tracking (TF-specific)
48 |        self._self setattr_tracking = _self setattr_tracking

```

Figure 7.10: File 2 - Before Refactoring

```

1 | def __init__(self):
2 |     if hasattr(self, "_tracker"):
3 |         return
4 |     trainable_variables = []
5 |     non_trainable_variables = [] You, 39 seconds ago + Uncommitted change
6 |     layers = []
7 |     metrics = []
8 |     seed_generators = []
9 |     self._tracker = tracking.Tracker(
10 |        {
11 |            "trainable_variables": [
12 |                lambda x: isinstance(x, backend.Variable) and x.trainable,
13 |                trainable_variables,
14 |            ],
15 |            "non_trainable_variables": [
16 |                lambda x: isinstance(x, backend.Variable)
17 |                and not x.trainable,
18 |                non_trainable_variables,
19 |            ],
20 |            "metrics": (lambda x: isinstance(x, Metric), metrics),
21 |            "layers": [
22 |                lambda x: isinstance(x, Layer)
23 |                and not isinstance(x, Metric),
24 |                layers,
25 |            ],
26 |            "seed_generators": [
27 |                lambda x: isinstance(x, backend.random.SeedGenerator),
28 |                seed_generators,
29 |            ],
30 |        }
31 |    )
32 |
33 |    if backend.backend() == "tensorflow":
34 |        # Remove attribute tracking for lists (TF-specific attribute)
35 |        _self setattr_tracking = getattr(
36 |            self, "_self setattr_tracking", True
37 |        )
38 |        self._self setattr_tracking = False
39 |        self._trainable_variables = trainable_variables
40 |        self._non_trainable_variables = non_trainable_variables
41 |        self._layers = layers
42 |        self._metrics = metrics
43 |        self._seed_generators = seed_generators
44 |    if backend.backend() == "tensorflow":
45 |        # Reset attribute tracking (TF-specific)
46 |        self._self setattr_tracking = _self setattr_tracking

```

Figure 7.11: File 2 - After Refactoring

After executing our tool on the file shown in figure 7.10 and 7.11 it is seen that although the code has small if statements it does not refactor the code. However, the indentations and the empty lines are removed by the `check_file_format.py` file. Our design makes sure that if there is not any if-else statement that is between six to fifty lines the tool will ignore it.

7.2.3 File 3: Complex Example - I

Our main goal in using this complex example was to find out how our tool performed when it came to code bases that had sophisticated python syntax. Moreover, using these examples helped us in finding edge cases and we tried to resolve as many edge cases we could. However, the most challenging part for us in this example was to find out the parameters that should have been passed into the method that our algorithm extracted.

```

1 def _real_extract(self, url):
2     video_id = self._match_id(url).split('/')[-1]
3     def call_playback_api(item, query=None):
4         try:
5             return self._call_api('playback/[%item%]/program/%video_id%', video_id)
6         except ExtractorError as e:
7             if isinstance(e.cause, HTTPError) and e.cause.status == 404:
8                 return self._call_api('playback/[%item%]/compat_str' if query else None)
9             raise
10    # Known values for preferredlangs: akadn, iapnly, ...
11    manifest = call_playback_api('manifest', {'pref': 'View Problem (VFB) Quick Fix'})
12    video_id = try_get(manifest, lambda x: x['id'], compat_str) or video_id
13    if manifest.get('playability') == 'nonPlayable':
14        self._raise_error(manifest['nonPlayable'])
15    playable = manifest['playable']
16    formats = []
17    for asset in playable['assets']:
18        if not isinstance(asset, dict):
19            continue
20        if asset.get('encrypted'):
21            continue
22        format_url = url_or_none(asset.get('url'))
23        if not format_url:
24            continue
25        asset_format = (asset.get('format') or '').lower()
26        if asset_format == 'hlsl' or determine_ext(format_url) == 'm3u8':
27            formats.extend(self._extract_nrk_formats(format_url, video_id))
28        elif asset_format == 'mp3':
29            formats.append({
30                'url': format_url,
31                'format_id': asset_format,
32                'vcodec': 'none',
33            })
34    data = call_playback_api('metadata')
35    preplay = data['preplay']
36    titles = preplay['titles']
37    title = titles['title']
38    alt_title = titles.get('subtitle')
39    description = try_get(preplay, lambda x: x['description'].replace('\r', '\n'))
40    duration = parse_duration(playable.get('duration')) or parse_duration(data['duration'])
41    thumbnails = []
42    for image in try_get(
43        preplay, lambda x: x['poster']['images'], list) or []:
44        if not isinstance(image, dict):
45            continue
46        image_url = url_or_none(image.get('url'))
47        if not image_url:
48            continue
49        thumbnails.append({
50            'url': image_url,
51            'width': int_or_none(image.get('pixelWidth')),
52            'height': int_or_none(image.get('pixelHeight')),
53        })
54    subtitles = {}
55    for sub in try_get(playable, lambda x: x['subtitles'], list) or []:
56        if not isinstance(sub, dict):
57            continue
58        sub_url = url_or_none(sub.get('webVtt'))
59        if not sub_url:
60            continue
61        sub_key = str_or_none(sub.get('language')) or 'nb'
62        sub_type = str_or_none(sub.get('type'))
63        if sub_type:
64            sub_key += '-' + sub_type
65        subtitles.setdefault(sub_key, []).append({
66            'url': sub_url,
67        })
68    legal_age = try_get(
69        data, lambda x: x['legalAge'][('body')[('rating')[('code')]])], compat_str)
70    # https://en.wikipedia.org/wiki/Norwegian_Media_Authority
71    age_limit = None
72    if legal_age:
73        if legal_age == 'A':
74            age_limit = 0
75        elif legal_age.isdigit():
76            age_limit = int_or_none(legal_age)
77    is_series = try_get(data, lambda x: x['links'][('series')][('name')]) == 'series'
78    info = {
79        'id': video_id,
80        'title': title,
81        'alt_title': alt_title,
82        'description': description,
83        'duration': duration,
84        'thumbnails': thumbnails,
85        'age_limit': age_limit,
86        'formats': formats,
87        'subtitles': subtitles,
88        'timestamp': parse_iso8601(try_get(manifest, lambda x: x['availability']))
89    }
90    if is_series:
91        series = season_id = season_number = episode = episode_number = None
92        programs = self._call_api(
93            'programs/%s' % video_id, video_id, 'programs', fatal=False)
94        if programs and isinstance(programs, dict):
95            series = str_or_none(programs.get('seriesTitle'))
96
97    season_id = str_or_none(programs.get('seasonId'))
98    season_number = int_or_none(programs.get('seasonNumber'))
99    episode = str_or_none(programs.get('episodeTitle'))
100   episode_number = int_or_none(programs.get('episodeNumber'))
101
102   if not series:
103       series = title
104   if alt_title:
105       title += ' - %s' % alt_title
106   if not season_number:
107       season_number = int_or_none(self._search_regex(
108           r'(Season\d+)(\d+)', description or '', 'season number',
109           default=None))
110
111   if not episode:
112       episode = alt_title if is_series else None
113   if not episode_number:
114       episode_number = int_or_none(self._search_regex(
115           r'^(\d+)(\d+)', episode or '', 'episode number',
116           default=None))
117
118   if not episode_number:
119       episode_number = int_or_none(self._search_regex(
120           r'\((\d+)\)(\d+)', description or '',
121           'episode number', default=None))
122
123   info.update({
124       'title': title,
125       'series': series,
126       'season_id': season_id,
127       'season_number': season_number,
128       'episode': episode,
129       'episode_number': episode_number,
130   })
131
132   print('afslaf askifla addad', video_id)
133
134   return info

```

Figure 7.12: File 3 - Before Refactoring

```

1| from new_method import *
2| def _real_extract(self, url):
3|     video_id = self._match_id(url).split('/')[-1]
4|     def call_playback_api(item, query=None):
5|         try:
6|             return self._call_api("playback/{item}/program/{video_id}", video_id, item, query=query)
7|         except ExtractorError as e:
8|             if isinstance(e.cause, HTTPError) and e.cause.status == 400:
9|                 return self._call_api("playback/{item}/{video_id}", video_id, item, query=query)
10|             raise
11|     # Known values for preferredCdn: akamai, ipony, minicdn and telenor
12|     manifest = call_playback_api("manifest", {"preferredCdn": "akamai"})
13|     video_id = try_get(manifest, lambda x: x["id"], compat_str) or video_id
14|     if manifest.get("playability") == "nonPlayable":
15|         self._raise_error(manifest["nonPlayable"])
16|     playable = manifest["playable"]
17|     formats = []
18|     for asset in playable["assets"]:
19|         if not isinstance(asset, dict):
20|             continue
21|         if asset.get("encrypted"):
22|             continue
23|         format_url = url_or_none(asset.get("url"))
24|         if not format_url:
25|             continue
26|         asset_format = (asset.get("format") or "").lower()
27|         method_1(asset_format)
28|         data = call_playback_api("metadata")
29|         preplay = data["preplay"]
30|         titles = preplay["titles"]
31|         title = titles["title"]
32|         alt_title = titles.get("subtitle")
33|         description = try_get(preplay, lambda x: x["description"].replace("\r", "\n"))
34|         duration = parse_duration(playable.get("duration")) or parse_duration(data.get("duration"))
35|         thumbnails = []
36|         for image in try_get(
37|             preplay, lambda x: x["poster"]["images"], list
38|         ) or []:
39|             if not isinstance(image, dict):
40|                 continue
41|             image_url = url_or_none(image.get("url"))
42|             if not image_url:
43|                 continue
44|             thumbnails.append({
45|                 "url": image_url,
46|                 "width": int_or_none(image.get("pixelWidth")),
47|                 "height": int_or_none(image.get("pixelHeight")),
48|             })
49|     subtitles = {}
50|     for sub in try_get(playable, lambda x: x["subtitles"], list) or []:
51|         if not isinstance(sub, dict):
52|             continue
53|         sub_url = url_or_none(sub.get("webVtt"))
54|         if not sub_url:
55|             continue
56|         sub_key = str_or_none(sub.get("language")) or "id"
57|         sub_type = str_or_none(sub.get("type"))
58|         if sub_type:
59|             sub_key += "-%s" % sub_type
60|         subtitles.setdefault(sub_key, []).append({
61|             "url": sub_url,
62|         })
63|     legal_age = try_get(
64|         data, lambda x: x["legalAge"]["body"]["rating"]["code"], compat_str
65|     )
66|     # https://en.wikipedia.org/wiki/Norwegian_Media_Authority
67|     age_limit = None
68|     if legal_age:
69|         if legal_age == "A":
70|             age_limit = 0
71|         elif legal_age.isdigit():
72|             age_limit = int_or_none(legal_age)
73|     is_series = try_get(data, lambda x: x["_links"]["series"]["name"]) == "series"
74|     info = {
75|         "id": video_id,
76|         "title": title,
77|         "alt_title": alt_title,
78|         "description": description,
79|         "duration": duration,
80|         "thumbnails": thumbnails,
81|         "age_limit": age_limit,
82|         "formats": formats,
83|         "subtitles": subtitles,
84|         "timestamp": parse_isob8601(try_get(manifest, lambda x: x["availability"]["onDemand"])[0], str)
85|     }
86|     method_2(series, season_id, season_number, episode, episode_number, programs, series, title, description, episode, alt_title, is_series)
87|     return info
88| 
```

Figure 7.13: File 3 - After Refactoring

The three important observations that we analysed after applying our tool on file shown in figure 7.12 are:

1. Our tool was successful in finding out the portion of code that needed refactoring. Thus, our algorithm is well built, even for complex methods, to find out segments of code that need refactoring.
2. As discussed previously, our algorithm is still in development and it cannot handle parameters. It is evident as two of the methods that our tool extracted are either missing some parameters that should have been passed or extra parameters were passed leading to another sort of code smell of Long Parameter List as stated in methodology 5.1.
3. After undergoing refactoring it is seen that the line number decreased from 128 to 87.

Nevertheless, it is clear that long methods can be refactored effortlessly, which can be seen from figure 7.13, only if we can identify the part of code that needs to be extracted. Our naive approach can be evolved to handle such cases.

7.2.4 File 4: Complex Example - II

The main difference in this file, figure 7.14, compared to the previous one is that it contains a nested for-loop. Moreover, in the nested for-loop there is long if-elif statement with comments in between the if and elif statements. In addition, the code contains a comment that does not use the conventional hashtag. All these complexities produced a lot of challenges and helped us in finding the limitations of our algorithm.

Figure 7.14: File 4 - Before Refactoring

```
1 | from new_method import *
2 | def _parseBurpLog(content):
3 |     ...
4 |     Parses Burp logs
5 |     ...
6 |     method_1(content, reqResList, match, port, request, host)
7 |     for match in reqResList:
8 |         request = match if isinstance(match, six.string_types) else match.group(1)
9 |         request = re.sub(r'^A[^\\w]+', "", request)
10 |         schemePort = re.search("(http|https)://[^/]*.*?:(\d+)", request, re.I | re.S)
11 |         method_2(schemePort, scheme, port, request, scheme, request)
12 |         getPostReq = False
13 |         forceBody = False
14 |         url = None
15 |         host = None
16 |         method = None
17 |         data = None
18 |         cookie = None
19 |         params = False
20 |         newline = None
21 |         lines = request.split("\n")
22 |         headers = []
23 |         for index in range(len(lines)):
24 |             line = lines[index]
25 |             if not line.strip() and index == len(lines) - 1:
26 |                 break
27 |             newline = "\r\n" if line.endswith("\r") else "\n"
28 |             line = line.strip("\r\n")
29 |             match = re.search("^(A-Z)+ ([A-Z]+)( HTTP/(\d.)+Z)", line) if not method else None
30 |             method_3(method, data, params, match, url, line, getPostReq)
31 |             # POST parameters
32 |             elif data is not None and params:
33 |                 data += "%%%" % (line, newline)
34 |             # GET parameters
35 |             elif "?" in line and "=" in line and ":" not in line:
36 |                 params = True
37 |             # Headers
38 |             method_4(key, value, match, match_, cookie, scheme, port, port_, host, forceBody, params, key)
39 |             data = data.rstrip("\r\n") if data else data
40 |             if getPostReq and (params or cookie or not checkParams):
41 |                 method_5(port, scheme, host, errMsg, url, url, method, data, cookie)
42 |             content = readCachedFileContent(refFile)
43 |             method_6(regular, expression, ex, errMsg)
44 |             for target in parseBurpLog(content):
45 |                 yield target
46 |             for target in parseWebScarabLog(content):
47 |                 yield target
```

Figure 7.15: File 4 - After Refactoring

Along with the limitations of the tool that were discussed previously this file introduced us with additional problems, which can be seen on figure 7.15.

1. If there are continuous if-elif statements our technique in extracting the part of the code fails.
2. Our tool extracted out a method from line 28 to 38, which it should not have.
3. Also, it missed to refactor two elif statement that were present between line 66 and 72. The algorithm instead extracted out two methods: method_3 and method_4.
4. Method_5 should have also contained the if statement that is in line 40.
5. Method_6 contains python keywords as parameters.

Our algorithm performed poorly in this example and showed us that there are growing number of edge cases. Although our algorithm failed, we believe that if given time our tool would evolve to handle numerous edge cases and make the algorithm more dynamic. The snippet of the refactored file is given in the previous page.

Chapter 8

Conclusion

8.1 Limitations

In our research we were able to detect code smells using ensemble technique and neural network and also proposed a naive approach to refactoring one of the code smells, long method. However, there are limitations to our work that need addressing. Firstly, developers might not address some of the code smells that we proposed because they might find it difficult to address the code smells or the code smells have little to no effect on the code [23]. Secondly, our refactoring algorithm finds any long if-else statement and extracts it for refactoring. However, developers and software engineers have their own perspective on the segment of code that needs refactoring [18]. Addressing perspective is out of scope for our research. Lastly, our algorithm was not able to handle some of the files as discussed in the previous section and also the design of the algorithm is simple which gives rise to poor performance. The limitations of the algorithm are divided into two subsections, which is discussed below.

8.1.1 Incomplete Code Refactoring

In the previous section we have analysed in-depth, which kind of code our algorithm cannot handle and why it cannot handle it. In addition to the discussed limitations, there are a few more flaws that have not been addressed yet.

It is evident that a long for-loop causes a long method smell. Our tool cannot find the for-loop from given code and refactor it. Moreover, to refactor a python file we import the extracted method from another file. As the extracted method is written in another file it is unaware of any functions that are inside it. Thus, error occurs when we call these extracted methods in place of the actual code.

8.1.2 Performance Constraints

Our straightforward approach in designing the algorithm gives rises to heavy computations which increases the runtime of the tool if the python files are large. To solve the edge cases we used a lot of “if” statements, which requires a lot of computation. Moreover, our algorithm finds parts of code that are between a threshold and refactors it. However, this simplistic approach does not take any other factors

into consideration thus solves the problem using a greedy approach. The greediness causes the time complexity of the algorithm to be between linear and polynomial.

8.2 Future Work

After heavy investigation, we still feel that there is lots of room for improvement for our naive approach in refactoring the Long Method. As our future prospects, we would work on improving and mitigating on some of the aspects our algorithm like:

- Our algorithm still lacks in identifying loop statements in long methods which are one of the important codes in a long method that exhibits strong explanation of the overall method. Apart from identification of loop statements, we look forward to modifying the system in such a way so that it can also extract a chunk of loop statement from the input long method and refactor like it does for conditional statements.
- As discussed in the analysis portion of 7.2.3 and 7.2.4, the system fails to take care of complex parameters of the long methods, we would like to bring refinement in this part of the system as well. Therefore, without any human intervention it should be capable of adding the necessary parameters in the newly born functions and avoids taking any extra non-necessary parameters.
- It is evident that our tool can refactor simple Long methods but it is safe to say that it is quite computationally expensive and static in refactoring. As our future objectives, with the aid of the research community we would like to deeply investigate and put into practice the use of large language models so that these models are victorious in refactoring long methods. Large language models will be trained on billions of parameters, in order to teach the model on what a long method is and how to refactor a long method using extract method. As a result, our large language model can easily, dynamically, and with less computation it can produce a refactored version of our input Long Method.

8.3 Conclusion

Addressing the critical need for automated code smell detection and refactoring in maintaining software quality and bolstering maintainability, this paper focuses on Python, an area overlooked in prior research. Leveraging the PySmell tool [16] from the literature, we present a comprehensive multi-labeled code smell dataset tailored for compatibility with machine learning models. Demonstrating the efficacy of artificial neural networks and ensemble models in accurately detecting code smells, we extend our investigation to the next frontier – automated refactoring of long method using extract method. Employing a basic extract method algorithm, we aim to refactor identified long if-else statement inside a long method, subsequently analysing the results in decrease of code smell in four python files and finding the limitations of our unsophisticated algorithm. Our endeavor underscores the importance of preserving code cleanliness by detecting code smells that requires refactoring and providing a naive approach for refactoring to ensure optimal maintainability in Python code bases.

Bibliography

- [1] A. Lakhotia and J.-C. Deprez, “Restructuring programs by tucking statements into functions,” *Information and Software Technology*, vol. 40, no. 11, pp. 677–689, 1998, ISSN: 0950-5849. DOI: [https://doi.org/10.1016/S0950-5849\(98\)00091-3](https://doi.org/10.1016/S0950-5849(98)00091-3). [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0950584998000913>.
- [2] M. Drozdz, D. G. Kourie, B. W. Watson, and A. Boake, “Refactoring tools and complementary techniques.,” *AICCSA*, vol. 6, pp. 685–688, 2006.
- [3] A. C. P. L. F. de Carvalho and A. A. Freitas, “A tutorial on multi-label classification techniques,” in *Foundations of Computational Intelligence Volume 5: Function Approximation and Classification*, A. Abraham, A.-E. Hassanien, and V. Snášel, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 177–195, ISBN: 978-3-642-01536-6. DOI: 10.1007/978-3-642-01536-6_8. [Online]. Available: https://doi.org/10.1007/978-3-642-01536-6_8.
- [4] L. Yang, H. Liu, and Z. Niu, “Identifying fragments to be extracted from long methods,” in *2009 16th Asia-Pacific Software Engineering Conference*, 2009, pp. 43–49. DOI: 10.1109/APSEC.2009.20.
- [5] P. Larsen, R. Ladelsky, S. Karlsson, and A. Zaks, “Compiler driven code comments and refactoring,” in *Fourth Workshop on Programmability Issues for Multi-Core Computers (MULTIPROG-2011)*, Citeseer, 2011, p. 64.
- [6] S. R. Foster, W. G. Griswold, and S. Lerner, “Witchdoctor: Ide support for real-time auto-completion of refactorings,” in *2012 34th International Conference on Software Engineering (ICSE)*, 2012, pp. 222–232. DOI: 10.1109/ICSE.2012.6227191.
- [7] P. Larsen, R. Ladelsky, J. Lidman, S. A. McKee, S. Karlsson, and A. Zaks, “Parallelizing more loops with compiler guided refactoring,” in *2012 41st International Conference on Parallel Processing*, IEEE, 2012, pp. 410–419.
- [8] F. A. Fontana, M. Zanoni, A. Marino, and M. V. Mäntylä, “Code smell detection: Towards a machine learning-based approach,” in *2013 IEEE International Conference on Software Maintenance*, 2013, pp. 396–399. DOI: 10.1109/ICSM.2013.56.
- [9] A. Hamid, M. Ilyas, M. Hummayun, and A. Nawaz, “A comparative study on code smell detection tools,” *International Journal of Advanced Science and Technology*, vol. 60, pp. 25–32, 2013.
- [10] H. K. Wright, D. Jasper, M. Klimek, C. Carruth, and Z. Wan, “Large-scale automated refactoring using clangmr,” in *2013 IEEE International Conference on Software Maintenance*, IEEE, 2013, pp. 548–551.

- [11] B. Amal, M. Kessentini, S. Bechikh, J. Dea, and L. B. Said, “On the use of machine learning and search-based software engineering for ill-defined fitness function: A case study on software refactoring,” in *Search-Based Software Engineering: 6th International Symposium, SSBSE 2014, Fortaleza, Brazil, August 26-29, 2014. Proceedings 6*, Springer, 2014, pp. 31–45.
- [12] J. Read, A. Puurula, and A. Bifet, “Multi-label classification with meta-labels,” in *2014 IEEE International Conference on Data Mining*, 2014, pp. 941–946. DOI: 10.1109/ICDM.2014.38.
- [13] F. Arcelli Fontana, M. Mäntylä, M. Zanoni, and A. Marino, “Comparing and experimenting machine learning techniques for code smell detection,” *Empirical Software Engineering*, vol. 21, Jun. 2015. DOI: 10.1007/s10664-015-9378-4.
- [14] F. A. Fontana, M. Mangiacavalli, D. Pochiero, and M. Zanoni, “On experimenting refactoring tools to remove code smells,” in *Scientific Workshop Proceedings of the XP2015*, 2015, pp. 1–8.
- [15] G. S. ke, C. Nagy, L. J. Fülop, R. Ferenc, and T. Gyimóthy, “Faultbuster: An automatic code smell refactoring toolset,” in *2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2015, pp. 253–258. DOI: 10.1109/SCAM.2015.7335422.
- [16] Z. Chen, L. Chen, W. Ma, and B. Xu, “Detecting code smells in python programs,” in *2016 International Conference on Software Analysis, Testing and Evolution (SATE)*, 2016, pp. 18–23. DOI: 10.1109/SATE.2016.10.
- [17] E. Fernandes, J. Oliveira, G. Vale, T. Paiva, and E. Figueiredo, “A review-based comparative study of bad smell detection tools,” in *Proceedings of the 20th international conference on evaluation and assessment in software engineering*, 2016, pp. 1–12.
- [18] 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*, vol. 43, no. 10, pp. 954–974, 2017. DOI: 10.1109/TSE.2016.2645572.
- [19] F. A. Fontana and M. Zanoni, “Code smell severity classification using machine learning techniques,” *Knowledge-Based Systems*, vol. 128, pp. 43–58, 2017.
- [20] T. Paiva, A. Damasceno, E. Figueiredo, and C. Sant’Anna, “On the evaluation of code smells and detection tools,” *Journal of Software Engineering Research and Development*, vol. 5, pp. 1–28, 2017.
- [21] S. Sharma, S. Sharma, and A. Athaiya, “Activation functions in neural networks,” *Towards Data Sci*, vol. 6, no. 12, pp. 310–316, 2017.
- [22] Z. Chen, L. Chen, W. Ma, X. Zhou, Y. Zhou, and B. Xu, “Understanding metric-based detectable smells in python software: A comparative study,” *Information and Software Technology*, vol. 94, pp. 14–29, 2018, ISSN: 0950-5849. DOI: <https://doi.org/10.1016/j.infsof.2017.09.011>. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0950584916301690>.

- [23] Z. Chen, L. Chen, W. Ma, X. Zhou, Y. Zhou, and B. Xu, “Understanding metric-based detectable smells in python software: A comparative study,” *Information and Software Technology*, vol. 94, pp. 14–29, 2018, ISSN: 0950-5849. DOI: <https://doi.org/10.1016/j.infsof.2017.09.011>. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0950584916301690>.
- [24] D. Di Nucci, F. Palomba, D. A. Tamburri, A. Serebrenik, and A. De Lucia, “Detecting code smells using machine learning techniques: Are we there yet?” In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2018, pp. 612–621. DOI: 10.1109/SANER.2018.8330266.
- [25] M. Fowler, *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018.
- [26] N. Tsantalis, T. Chaikalis, and A. Chatzigeorgiou, “Ten years of jdeodorant: Lessons learned from the hunt for smells,” in *2018 IEEE 25th international conference on software analysis, evolution and reengineering (SANER)*, IEEE, 2018, pp. 4–14.
- [27] L. Kumar, S. M. Satapathy, and L. B. Murthy, “Method level refactoring prediction on five open source java projects using machine learning techniques,” ser. ISEC ’19, ¡conf-loc¿, ¡city¿Pune¿/city¿, ¡country¿India¿/country¿, ¡/conf-loc¿: Association for Computing Machinery, 2019, ISBN: 9781450362153. DOI: 10.1145/3299771.3299777. [Online]. Available: <https://doi.org/10.1145/3299771.3299777>.
- [28] M. Agnihotri and A. Chug, “A systematic literature survey of software metrics, code smells and refactoring techniques,” *Journal of Information Processing Systems*, vol. 16, no. 4, pp. 915–934, 2020.
- [29] M. Alenezi, M. Akour, and O. Al Qasem, “Harnessing deep learning algorithms to predict software refactoring,” *TELKOMNIKA (Telecommunication Computing Electronics and Control)*, vol. 18, no. 6, pp. 2977–2982, 2020.
- [30] M. Aniche, E. Maziero, R. Durelli, and V. H. Durelli, “The effectiveness of supervised machine learning algorithms in predicting software refactoring,” *IEEE Transactions on Software Engineering*, vol. 48, no. 4, pp. 1432–1450, 2020.
- [31] L. Aversano, U. Carpenito, and M. Iammarino, “An empirical study on the evolution of design smells,” *Information*, vol. 11, no. 7, 2020, ISSN: 2078-2489. DOI: 10.3390/info11070348. [Online]. Available: <https://www.mdpi.com/2078-2489/11/7/348>.
- [32] M. De Stefano, M. S. Gambardella, F. Pecorelli, F. Palomba, and A. De Lucia, “Casper: A plug-in for automated code smell detection and refactoring,” in *Proceedings of the International Conference on Advanced Visual Interfaces*, 2020, pp. 1–3.
- [33] J. Reis, F. Brito e Abreu, G. Carneiro, and C. Anslow, *Code smells detection and visualization: A systematic literature review*, Dec. 2020.
- [34] P. Sandrasegaran and S. Vengusamy, “Enhancing software quality using artificial neural networks to support software refactoring,” *EasyChair*, pp. 1–10, 2021.

- [35] Y. Tang, R. Khatchadourian, M. Bagherzadeh, R. Singh, A. Stewart, and A. Raja, “An empirical study of refactorings and technical debt in machine learning systems,” in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, 2021, pp. 238–250. doi: 10.1109/ICSE43902.2021.00033.
- [36] T. Wang, Y. Golubev, O. Smirnov, J. Li, T. Bryksin, and I. Ahmed, “Pynose: A test smell detector for python,” in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2021. doi: 10.1109/THS.2011.6107909.
- [37] S. Dewangaan, R. S. Rao, A. Mishra, and M. Gupta, “Code smell detection using ensemble machine learning algorithms,” Oct. 2022. doi: 10.3390/app122010321.
- [38] A. Kovačević, J. Slivka, D. Vidaković, *et al.*, “Automatic detection of long method and god class code smells through neural source code embeddings,” *Expert Systems with Applications*, vol. 204, p. 117607, 2022.
- [39] M. Shahidi, M. Ashtiani, and M. Zakeri-Nasrabadi, “An automated extract method refactoring approach to correct the long method code smell,” *Journal of Systems and Software*, vol. 187, p. 111221, 2022.
- [40] B. K. Sidhu, K. Singh, and N. Sharma, “A machine learning approach to software model refactoring,” *International Journal of Computers and Applications*, vol. 44, no. 2, pp. 166–177, 2022.
- [41] R. Sandouka and H. Aljamaan, “Python code smells detection using conventional machine learning models,” *Empirical Software Engineering*, May 2023. doi: 10.7717/peerj-cs.1370/supp-1.
- [42] M. Zakeri-Nasrabadi, S. Parsa, E. Esmaili, and F. Palomba, “A systematic literature review on the code smells datasets and validation mechanisms,” *ACM Computing Surveys*, vol. 55, no. 13s, pp. 1–48, 2023.
- [43] D. Al-Fraiha, Y. Sharrab, A.-R. Al-Ghuwairi, M. AlElaimat, and M. Alzaidi, “Detecting and resolving feature envy through automated machine learning and move method refactoring,” *International Journal of Electrical and Computer Engineering (IJECE)*, vol. 14, no. 2, pp. 2330–2343, 2024.
- [44] B. Nyirongo, Y. Jiang, H. Jiang, and H. Liu, “A survey of deep learning based software refactoring,” *arXiv preprint arXiv:2404.19226*, 2024.