

# Enhancing Software Quality: A Machine Learning Approach to Python Code Smell Detection and Refactoring

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  
September 2023

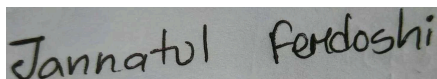
© 2023. Brac University  
All rights reserved.

# 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.

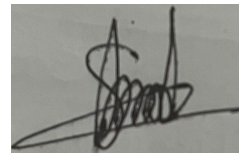
## Student's Full Name & Signature:



---

Jannatul Ferdoshi

20301193



---

Shabab Abdullah

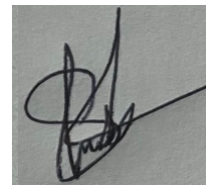
20301005



---

Kazi Zunayed Quader Knobo

20241020



---

Mohammed Sharraf Uddin

20241018

# Approval

Supervisor:

Md. Aquib Azmain

---

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

# Abstract

Automatic Program Repair (APR) has emerged as a promising field in software engineering, aiming to alleviate the burden of manual bug fixing by automatically generating patches for software defects. APR explores the use of program analysis, machine learning, and search-based algorithms to identify and fix software bugs. Various repair strategies, such as code synthesis, mutation-based repair, and constraint solving, are examined, highlighting their strengths and limitations. The second part of the abstract focuses on recent advancements in APR. It discusses the emergence of deep learning approaches, including neural networks and reinforcement learning, to enhance repair techniques. These techniques leverage vast amounts of code repositories and bug databases to learn patterns and generate more accurate and context-aware fixes. In conclusion, automatic program repair holds great promise for software development by reducing the manual effort and time required for bug fixing. Ultimately, the continued advancement of APR can significantly enhance software quality and developer productivity in the years to come.

**Keywords:** Automated Program Repair, Code Smells, Code Refactoring, Machine Learning, Code Analysis, Software Maintenance, GitHub Repositories

# Table of Contents

<b>Declaration</b>	<b>i</b>
<b>Approval</b>	<b>ii</b>
<b>Abstract</b>	<b>iii</b>
<b>Table of Contents</b>	<b>iv</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Background . . . . .	1
1.2 Problem Statement . . . . .	2
1.3 Research Objective . . . . .	2
<b>2 Detailed Literature Review</b>	<b>3</b>
2.1 PyNose: A Test Smell Detector For Python . . . . .	3
2.2 FaultBuster: An automatic code smell refactoring toolset . . . . .	4
2.3 Detecting Code Smells in Python Programs . . . . .	4
2.4 Code Smell Detection: Towards a Machine Learning-Based Approach	5
2.5 Detecting code smells using machine learning techniques: Are we there yet? . . . . .	6
2.6 Code smells detection and visualization: A systematic literature review	7
2.7 Understanding metric-based detectable smells in Python software: A comparative study . . . . .	8
2.8 Comparing and experimenting machine learning techniques for code smell detection . . . . .	9
2.9 Python code smells detection using conventional machine learning models . . . . .	10
2.10 Code Smell Detection Using Ensemble Machine Learning Algorithms	11
<b>3 Work Plan</b>	<b>13</b>
<b>4 Conclusion</b>	<b>14</b>
<b>Bibliography</b>	<b>15</b>

# 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. 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. These problems show up as unnecessary code, excessively complicated structures, or poor design decisions. 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 smell 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 a software.

The world of software development is changing dramatically as technology continues to improve at an astounding rate. A proactive approach to maintainability is required for software development in the future, and automated tools and approaches are well-positioned to play a key role. The ever-increasing complexity of software systems necessitates the identification and correction of code smells. The stakes are significant at a time when software runs everything from driverless automobiles to healthcare services. It is crucial to have code that is not just functional but also tidy, adaptive, and maintainable. By utilizing automated methods, we can speed up software recovery, improve codebase health, and make sure that future software is durable and adaptive to changing technological needs.

## 1.2 Problem Statement

While building softwares it is important to ensure that it is maintained, it performs well and has a high longevity. This can be done through identification and remediation of code smells in a software. Despite code smell detection and refactoring being one of the fundamental factors for writing clean code, however, there lies 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 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: How to design a python dataset that would allow machine learning algorithms to effectively analyze and recognize code smells?
- RQ2: How to train the machine learning algorithms in order to detect code smells from the python dataset?
- RQ3: How to create an intelligent system that would provide actionable refactoring suggestions?

## 1.3 Research Objective

This paper aims to develop an automated detection and refactoring tool for Python projects. The .py files will be extracted from GitHub repositories containing Python projects by a file analyzer. The tool in discussion will be able to automatically detect five common code smells in the extracted .py files, including:

- Duplicate code segments.
- Excessive or irrelevant comments.
- Long and complex methods.
- Instances of feature envy, where one module excessively relies on the internal details of another.
- Code blobs or large, monolithic code structures

Furthermore, the paper will discuss how different refactoring algorithms were implemented inside the tool to refactor the detected code smells without human intervention, improving the quality of the code base.

We will then evaluate the effectiveness and efficiency of our tool by measuring metrics such as code duplication reduction, comment relevance improvement, method length reduction, reduced feature envy, and modularization of code blobs. Moreover, we will keep track of the percentage of code smell before and after automated refactoring to measure improvement in software quality after integrating the tool.

# Chapter 2

## Detailed Literature Review

### 2.1 PyNose: A Test Smell Detector For Python

The paper [8] 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 authors, Golubev et al., draw attention to a sizable gap in the testing tool landscape by pointing out that while tools for identifying test smells are readily available for Java and Scala, they are conspicuously absent for Python. The paper [8] describes how a brand-new tool called Pynose was created to fill this gap. The first step in their methodology entails choosing particular test smells for analysis [8]. In a systematic mapping study of test smells currently under investigation, Golubev et al. identified 33 different test smells in Java, Scala, and Android systems. A rigorous filtering process led to the retention of 17 test smells that were customized for Python's unittest testing framework. The paper also emphasizes how crucial it is for the framework of the tool to address Python-specific test smells.

Subsequently, Golubev et al. embarked on the creation of a primary dataset, meticulously curating 450 projects from GHTorrent, all meeting rigorous criteria: a minimum of 50 stars, no forks, 1000 commits, and at least 10 contributors. To pinpoint Python-specific test smells within this dataset, the authors employed PYTHON-CHANGEMINER, a tool adept at tracking changes made to test files [8]. Their investigation revealed issues such as the misuse of assert functions, leading to a particular test smell known as "suboptimal assert." Additionally, a secondary dataset comprising 239 projects was established to evaluate Pynose's precision and accuracy in identifying test smells.

In the final phase of their research, Golubev et al. designed and built the Pynose tool, ingeniously integrating it as a plugin for PyCharm. The tool's workflow commences with the parsing and analysis of Python source code using JetBrains' IntelliJ Platform's Python Structure Interface (PSI), ensuring both syntactic and semantic scrutiny [8]. Pynose then selectively extracts classes inheriting from `unittest.TestCase` employs specific detector classes to identify test smells within these extracted classes. The tool presents detected code smells within 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 [8].

## 2.2 FaultBuster: An automatic code smell refactoring toolset

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 [3]. 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 [3] 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 [3].

## 2.3 Detecting 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 [4] is to identify code smells and provide support for refactoring strategies, ultimately enhancing the software quality of Python programs.

Initially, Chen et al. confronted the task of cataloging code smells applicable to Python, acknowledging that certain conventional code smells may not be relevant

to the Python context. To discern the characteristic Python code smells, they conducted an exhaustive review of online resources and reference manuals. Noteworthy Python code smells encompassed Large Class, Long Parameter, Long Method, Long Message Chain, and Long Scope Chaining [4].

Subsequently, Chen et al. curated a dataset comprising five open-source Python systems, namely django, ipython, matplotlib, scipy, and numpy, constituting a substantial codebase encompassing 626,087 lines of code across 4,592 files.

Finally, Chen et al. introduced Pysmell, a tool designed to identify code smells based on relevant metrics. Pysmell’s architecture comprises three core components: i) The Code Extractor, which examines the Python system’s design and extracts pivotal Python files.

ii) The AST Analyzer, responsible for processing the extracted Python code, constructing an Abstract Syntax Tree (AST), and analyzing the AST comprehensively while collecting the requisite metrics.

iii) The Smell Detector, where code smells are pinpointed based on the metrics gathered during the previous stage.

Chen et al. conducted an evaluation of their tool, achieving an impressive precision rate of approximately 98% and a mean recall of 100% in the detection of code smells within Python systems. However, it’s worth noting that this detection relies solely on metrics, potentially introducing some biases despite the remarkable results observed during the evaluation [4].

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

The paper titled ”Code Smell Detection: Towards a Machine Learning-Based Approach” by Francesca Arcelli Fontana, Marco Zanoni, and Alessandro Marino [1] presents a research endeavor focused on the application of machine learning techniques for the detection of code smells in software development. Code smells are inefficient patterns that can harm the quality and maintainability of software. In software engineering, finding code smells is crucial to ensure the durability and readability of codebases. This paper discusses the difficulties in code smell detection and proposes a unique method that uses machine learning to enhance the precision and effectiveness of this procedure. The authors begin their investigation by recognizing the existence of multiple code scent detection methods, each of which produces a distinct set of findings since code smell interpretation is a subjective process. The primary focus of these tools is metrics computation, but they frequently overlook important contextual considerations like the domain, size, and design elements of the program being analyzed. Achieving consistent and trustworthy findings is difficult due to the differences in these tools’ metrics usage and threshold values established. Furthermore, a lot of tools generate a lot of false positives, which might result in pointless code restructuring. The authors suggest a machine learning-based method for code scent detection in response to these difficulties. They draw attention to how little machine learning approaches have been explored in this area and stress

the need for more reliable and precise detection strategies. The processes of their methodology are described in detail in the paper, and they involve data collection, code scent selection, application of detection techniques, manual labeling of code smell candidates, and machine learning classifier experimentation. The Qualitas Corpus is a broad collection of 76 software systems that has been enhanced with a wide range of object-oriented metrics, which the authors draw on. They choose code smells carefully, concentrating on those that are prevalent, have a big influence on software quality, and can be found with current techniques. Code smells like God Class, Data Class, Long Method, and Feature Envy are included in this list. Importantly, the authors provide a mechanism for categorizing code smells according to their intensity, enabling a complex assessment of their influence on software quality. This rating, which spans from "No smell" to "Severe smell," gives developers a framework for setting priorities for efficiently managing code smells. Inspection of potential code smell candidates and severity labeling based on observable code features are steps in the manual evaluation process. In order to train supervised machine learning classifiers, this human-generated labeled dataset is employed. To find the best method for detecting code smells, the authors test a range of classifiers, including Support Vector Machines, Decision Trees, Random Forest, Naive Bayes, JRip, and boosting techniques. A binary label indicating the presence or absence of a code scent and object-oriented metrics are aspects of the datasets for class-level and method-level code smells that are being created. By normalizing and standardizing the indicators, the authors take precautions to assure their comparability. The authors use k-fold cross-validation to measure the performance of the classifiers, resulting in learning curves that show how quickly the algorithms pick up new information and how well they perform overall. Furthermore, classifiers like J48, Random Forest, and JRip offer human-readable rules, illuminating the metrics most important in detecting code smells. The performance of machine learning techniques for identifying various code smells is shown in preliminary findings. For each sort of code smell, the accompanying tables show the accuracy, F-measure, and ROC Area scores. These first results show how machine learning has the potential to increase the precision of code smell detection.

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

A paper by Dario Di Nucci et al. [6] explores the application of machine learning (ML) techniques in the detection of code smells. Code smells can significantly affect the maintainability and quality of source code since they are signs of poor design choices. Dario Di Nucci et al. examine the difficulties in detecting code smells as well as the possible advantages of applying ML to this problem. The paper discusses the growing complexity of software systems, which frequently forces developers to choose between producing software fast and following sound programming principles. These compromises can lead to the accumulation of technical debt, such as code smells, which are poor design choices that make it difficult to maintain code. Various code smell detection algorithms have been created over time, however they all have drawbacks, according to Dario Di Nucci et al. They can be subjective, meaning that various tools may distinguish between different types of code smells, and they

frequently call for the establishment of thresholds, which can have an impact on how well they function. The study proposes the use of ML approaches for code smell detection as a solution to these thresholds. By using information derived from source code to train classifiers, machine learning (ML) offers a method to automatically detect code smells. Dario Di Nucci et al. concentrate particularly on supervised ML techniques, where the existence or severity of code smell in code components is assessed using independent variables (predictors). The research was carried out on a broad scale by Arcelli Fontana et al. to identify four different forms of code smells: Data Class, God Class, Feature Envy, and Long Method. The study’s findings were encouraging, with the majority of classifiers reaching accuracy and F-Measure rates above 95 percentage. The study’s authors came to the conclusion that ML algorithms are appropriate for detecting code smells and that the selection of an ML method might not have a substantial influence on the outcomes. Dario Di Nucci et al. express uncertainty regarding how far these results may be applied. They make the observation that the dataset may have an impact on the high performance mentioned by Arcelli Fontana et al. The original study had an unbalanced dataset since each dataset for a particular type of code smell had only examples impacted by that type of smell or non-smelly instances. The research gives a repeatable study to allay these worries. With a separate dataset comprising code elements impacted by various code smells, the authors rerun the study and investigate the metric distribution of smelly and non-smelly code elements. The distribution of smelly and non-smelly cases in this new dataset is less balanced and more closely mimics real-world events. According to the new research’s findings, the initial study’s impressive performance was really the product of the dataset utilized, not the underlying potential of machine learning models. Code smell prediction models were up to 90 percentage less accurate in terms of F-Measure when evaluated on the updated dataset than they had been when tested on the original dataset, which had dramatically different metric distributions of smelly and non-smelly parts.

## 2.6 Code smells detection and visualization: A systematic literature review

Another paper titled "Code Smells Detection and Visualization: A Systematic Literature Review" by Jose Pereira dos Reis, Fernando Brito e Abreu, Glauco de Figueiredo Carneiro, and Craig Anslow [7] conducts a systematic literature review (SLR) on the topic of code smell detection and visualization. This review investigates the various techniques and tools used for detecting code smells in software and explores the extent to which visualization techniques have been applied to support these detection methods. Code smells, commonly referred to as "bad smells," are problems with software design and code that lower program quality and make maintenance more difficult. The health and durability of software systems depend on the ability to recognize and correct code smells. The SLR seeks to shed light on cutting-edge methods and equipment for code smell visualization and detection. The authors start out by talking about the difficulties of manually detecting code smells, emphasizing how subjective this process is. Large codebases may make manual detection problematic, and different developers may have different interpretations of code smells. As a result, automated detection techniques are essential for effectively

locating and reducing 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 percentage), metric-based (24.1percentage), and symptom-based (19.3percentage). While metric-based techniques depend on software metrics to identify code smells, search-based approaches seek for 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 (77.1percentage) in studies that employ open-source tools for code smell detection.
3. **Common Code Smells:** God Class (51.8percentage), Feature Envy (33.7percentage), and Long Method (26.5percentage) 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 35percentage 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 80percentage 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 outlines issues with code smell detection, such as lowering subjectivity in defining and detecting code smells, increasing the variety of detected code smells and supported programming languages, and offering oracles and datasets to make it easier to validate code smell detection and visualization methods. As a result, this thorough literature evaluation offers insightful information about the state of code smell visualization and detection today. It emphasizes the value of automated detection techniques, particularly when working with extensive and intricate software systems. The paper also emphasizes the need for enhanced visualization approaches to assist practitioners in efficiently identifying and treating code smells. The results of this SLR provide a basis for further investigation into code scent detection and visualization, with the ultimate objective of raising the quality and maintainability of software.

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

Zhifei Chen et al. addresses the issue of code smells in Python. With its dynamic nature and simple syntax, python has received less code smell study than static

languages like Java and C-sharp.[5] The main objective of this study is to define and detect code smells specific to Python programs and explore their impact on software maintainability. The paper introduces ten code smells in Python and establishes a metric-based detection method using three different threshold-setting strategies: experience-based, statistics-based, and tuning machine. The study utilizes a corpus of 106 popular Python projects from GitHub to conduct a comparative analysis of these detection strategies. The literature analysis of the paper focuses on code smells as indicators of difficulties in software design and implementation. It highlights the importance of addressing code smells to improve the quality and maintainability of programs. The study presents algorithms for detecting code smells, including metric-based machine learning, history-based, and textual approaches. While metric-based detection is commonly used and effective, setting thresholds can be challenging. In Python code, smells can impact readability and maintainability due to its syntax and dynamic nature. However, defining baselines for Python code smell measures poses difficulties in determining thresholds. To tackle this issue, the paper employs three detection strategies to identify and quantify Python smells, providing insights into their prevalence and impact on software modules. The contributions of this study include introducing ten Python smells based on metrics, developing the Pysmell detection tool, and performing an evaluation of three detection strategies. The findings reveal that these strategies complement each other in understanding Python smells. Long methods and lengthy parameter lists are found to be prevalent among smells. Strongly correlated with module change proneness and fault proneness. Additionally, the study sheds light on variations in developers perceptions of Python smells.

## **2.8 Comparing and experimenting machine learning techniques for code smell detection**

Another research by Francesca Arcelli Fontana et al. focuses on the utilization of machine learning (ML) techniques to detect code smells – indicators of code or design issues.[2] Code smells have been a concern in software maintenance and quality enhancement. However their identification has proven challenging due to varying interpretations and the absence of rules. To address these challenges this study explores the application of ML methods for automating code smell detection. The paper discusses the challenges associated with identifying code smells highlighting their interpretation and the absence of measures or thresholds. It suggests that code smell detection should be more focused and less arbitrary, proposing the use of machine learning (ML) technology to enable detectors to learn from cases. The empirical experimentation conducted forms the core contribution of the paper. The methodology employed involves a dataset of software systems and a diverse range of ML algorithms. In this approach specific code smells like Data Class, God Class, Feature Envy and Long Method are considered as variables while independent variables consist of software design metrics. The paper emphasizes the importance of selecting and labeling example instances based on existing detection tools results 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 algorithms in terms of performance while support

vector machines exhibited lower results. The paper acknowledges that imbalanced data influenced algorithm performance due to the prevalence of code smells. However the research findings suggest that machine learning methods have the capability to achieve a level of accuracy in identifying code smells. Moreover it is noteworthy that even a limited number of training examples can result in an accuracy rate of 95

## 2.9 Python code smells detection using conventional machine learning models

Rana S. and Hamoud A. [10] made a clear statement on how identifying code smells at an early stage of software development is crucial to improve 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 language 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 have implemented machine learning algorithms to find the expected code smells and automate the whole process.

Rana S. and Hamoud A.[10] have created their own datasets containing Python code smells. This is done as Python is the most used language in creating Data Science and Machine Learning models. Therefore, they have designed datasets based on two criterias. 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 Java programming language. As a result, they have chosen Long Method and Large Class code smells for their Python datasets. In the creation of the Python code smell dataset, four python libraries like 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 such as 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: Matthews correlation coefficient (MCC) and Accuracy.

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

code smell.

## 2.10 Code Smell Detection Using Ensemble Machine Learning Algorithms

Seema et al. [9] discusses that code smells in a 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. [9] fulfilled this requirement through their research as they have considered all the subsets of metrics and applied the algorithms to each group of metrics and found its effects on the model's performance and its accuracy. Seema et al. [9] exposes code smells by building a model. At first, they figured out the datasets of code smells and applied min-max normalization in order to scale features. Then, 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 smells like God Class, Data Class, Long Method, and Feature Envy in 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 method level datasets were Feature envy and Long method. In order to rescale the feature values of the datasets between 0-1, 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 Chi-square based feature selection approach is 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 of 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 like Artificial neural network, and Convolutional neural network on 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 the 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.

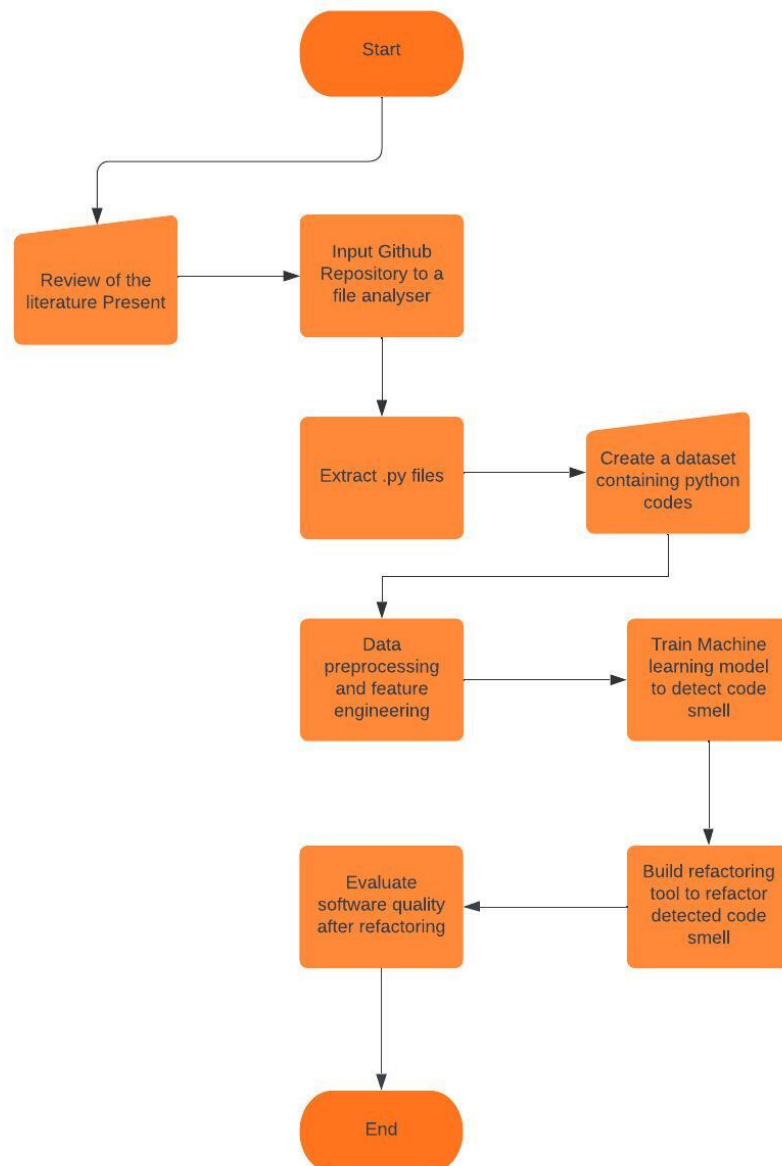
Decision tree algorithm surpassed all the other In the second half of their project Seema et al. [9] computed performance measures to compare the performance of machine learning techniques. The performance measurements are as follows, the



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 a software.

# Chapter 3

## Work Plan



# Chapter 4

## Conclusion

The exploration of code smell detection and automated program repair, in particular, illustrates the crucial significance these ideas play in the dynamic field of software development. We've looked at the fundamentals of code smell, how to recognize it as a crucial sign of future problems in codebases, and the significance of dealing with it to preserve code quality, cut maintenance costs, and avoid technological debt. The necessity for automated methods to eliminate code smell is obvious in the future, offering more durable and adaptive software systems in a time of fast technological innovation. We come to a deep understanding as we draw to a close this chapter: Automated Program Repair is more than just a tool; it is a ray of hope, pointing developers in the direction of painless program recovery and assuring that our software is flexible and robust in the face of upcoming obstacles. In the never-ending quest for excellence in software development, accepting this novel idea is not simply an option; it is a need.

# Bibliography

- [1] 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.
- [2] 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.
- [3] G. S. ke, C. Nagy, L. J. Fülöp, 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.
- [4] 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.
- [5] 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>.
- [6] 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.
- [7] J. Reis, F. Brito e Abreu, G. Carneiro, and C. Anslow, *Code smells detection and visualization: A systematic literature review*, Dec. 2020.
- [8] 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.
- [9] S. Dewangaan, R. S. Rao, A. Mishra, and M. Gupta, “Code smell detection using ensemble machine learning algorithms,” Oct. 2022. DOI: 10.3390/app122010321.
- [10] 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.