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

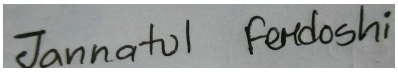
Department of Computer Science and Engineering
Brac University
September 2023

Declaration

It is hereby declared that

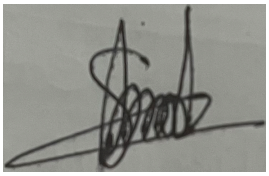
1. The thesis submitted is our own original work while completing degree at Brac University.
2. The thesis does not contain material previously published or written by a third party, except where this is appropriately cited through full and accurate referencing.
3. The thesis does not contain material 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:



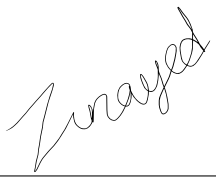
Jannatul Ferdoshi

20301193



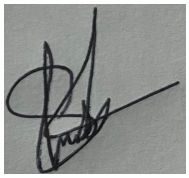
Shabab Abdullah

20301005



Kazi Zunayed Quader Knobo

20241020

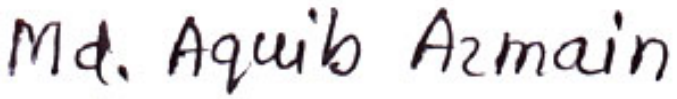


Mohammed Sharraf Uddin

20241018

Approval

Supervisor:



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

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 by constructing a dataset conducive to machine learning algorithms for effective code smell detection. Furthermore, our approach extends to automated refactoring, aiming to reduce code smells 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

Table of Contents

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. 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.

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 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: 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:

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

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 class length reduction, method length reduction, and reduction in the length of the message chain. 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 is 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] focuses on applying machine learning techniques to detect code smells in software development. Code smells are inefficient patterns that can negatively impact the quality and maintainability of software. Detecting code smells is essential in software engineering to ensure the durability and readability of codebases. The paper discusses the challenges in code smell detection and proposes a unique method that utilizes machine learning to enhance precision and effectiveness in this process.

The authors begin their investigation by acknowledging the existence of multiple code smell detection methods, each producing distinct findings. Code smell interpretation is subjective, leading to variations in results. Many existing tools focus on metrics computation but often overlook important contextual considerations such as domain, size, and design elements of the analyzed program. Inconsistencies arise from different tools' metric usage and threshold values, resulting in difficulties in achieving consistent and trustworthy findings. Additionally, false positives generated by some tools may lead to unnecessary code restructuring.

To address these challenges, the authors propose a machine learning-based method

for code smell detection. They highlight the limited exploration of machine learning approaches in this area and emphasize the need for more reliable and precise detection strategies. The paper details the methodology, including data collection, code smell selection, application of detection techniques, manual labeling of code smell candidates, and machine learning classifier experimentation. The authors leverage the Qualitas Corpus, a comprehensive collection of 76 software systems with object-oriented metrics.

The authors carefully choose prevalent and impactful code smells, such as God Class, Data Class, Long Method, and Feature Envy, and categorize them according to their intensity. This intensity rating, ranging from "No smell" to "Severe smell", provides developers with a framework for prioritizing code smell management effectively. The manual evaluation process involves inspecting potential code smell candidates and labeling their severity based on observable code features. The human-generated labeled dataset is then used to train supervised machine-learning classifiers. The authors test various classifiers, including Support Vector Machines, Decision Trees, Random Forest, Naive Bayes, JRip, and boosting techniques, to find the most effective method for detecting code smells. The results, presented in tables, demonstrate the potential of machine learning to enhance the precision of code smell detection, with accuracy, F-measure, and ROC Area scores for different types of code smells.

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

The paper by Dario Di Nucci et al.[6] delves into the application of machine learning (ML) techniques for detecting code smells, which are indicators of poor design choices that can significantly impact the maintainability and quality of source code. The authors acknowledge the challenges in code smell detection and discuss the potential advantages of leveraging ML to address these issues. They highlight the increasing complexity of software systems, forcing developers to make trade-offs between speedy software production and adhering to sound programming principles. Such compromises often lead to the accumulation of technical debt, including code smells, making code maintenance challenging.

The paper notes the drawbacks of existing code smell detection algorithms, including subjectivity and the need for establishing thresholds that can influence their effectiveness. In response to these challenges, the authors propose the use of ML approaches for code smell detection. By utilizing information derived from source code to train classifiers, ML offers an automated method for detecting code smells. The focus is particularly on supervised ML techniques, where the existence or severity of code smells in code components is assessed using independent variables (predictors).

The research conducted by Arcelli Fontana et al. serves as the foundation for the study, identifying four different forms of code smells: Data Class, God Class, Feature Envy, and Long Method. The initial findings were promising, with most classifiers achieving accuracy and F-Measure rates above 95 percent. The authors conclude that ML algorithms are suitable for detecting code smells, and the choice of an ML

method may not substantially impact the outcomes.

However, Dario Di Nucci et al. express uncertainty about the generalizability of these results. They raise concerns about the impact of the dataset on the high performance reported by Arcelli Fontana et al. The original study used an unbalanced dataset, with each dataset for a specific type of code smell having examples affected by that smell or non-smelly instances. To address these concerns, the authors conduct a repeatable study with a separate dataset comprising code elements impacted by various code smells. The new dataset has a less balanced distribution of smelly and non-smelly cases, more closely mimicking real-world scenarios. The updated research reveals that the initial study's impressive performance was attributed to the dataset used rather than the inherent potential of ML models. Code smell prediction models were up to 90 percent less accurate in terms of F-Measure when evaluated on the updated 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 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%), 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 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 specific to Python, emphasizing that Python, with its dynamic nature and simple syntax, has received less attention in code smell studies compared to static languages like Java and C-sharp[5]. The primary objective of this study is to define and detect code smells in Python programs and explore their impact on software maintainability. The paper introduces ten code smells specific to Python and establishes a metric-based detection method using three different threshold-setting strategies: experience-based, statistics-based, and tuning machine.

To conduct a comparative analysis of these detection strategies, the study utilizes a corpus of 106 popular Python projects sourced from GitHub. The literature analysis in the paper underscores the significance of addressing code smells as indicators of difficulties in software design and implementation, emphasizing their impact on the quality and maintainability of programs.

The study presents algorithms for detecting Python code smells, including metric-based machine learning, history-based, and textual approaches. While metric-based

detection is common and effective, the paper notes that setting thresholds can be challenging in Python due to its syntax and dynamic nature, which can impact readability and maintainability. To overcome this challenge, 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 evaluating three detection strategies. The findings reveal that these strategies complement each other in understanding Python smells, with long methods and lengthy parameter lists being prevalent among identified smells. The study also highlights their strong correlation with module change-proneness and fault-proneness. Furthermore, the research sheds light on variations in developers' perceptions of Python smells.

2.8 Comparing and experimenting machine learning techniques for code smell detection

Francesca Arcelli Fontana et al. presents a research study focusing on the application of machine learning (ML) techniques to detect code smells, which serve as indicators of code or design issues[2]. Code smells have been a persistent concern in software maintenance and quality improvement, yet their identification poses challenges due to varying interpretations and the absence of standardized rules. In response to these challenges, this study explores the use of ML methods to automate the detection of code smells.

The paper discusses the difficulties associated with identifying code smells, emphasizing issues related to interpretation and the lack of defined measures or thresholds. It advocates for a more focused and less arbitrary approach to code smell detection, proposing the integration of ML technology to enable detectors to learn from specific cases. The core contribution of the research lies in its empirical experimentation, utilizing a dataset of software systems and employing a diverse range of ML algorithms.

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. [10] 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.[10] 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), multilayer 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. [9] 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. [9] 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. [9] 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. [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 software.

Chapter 3

Work Plan

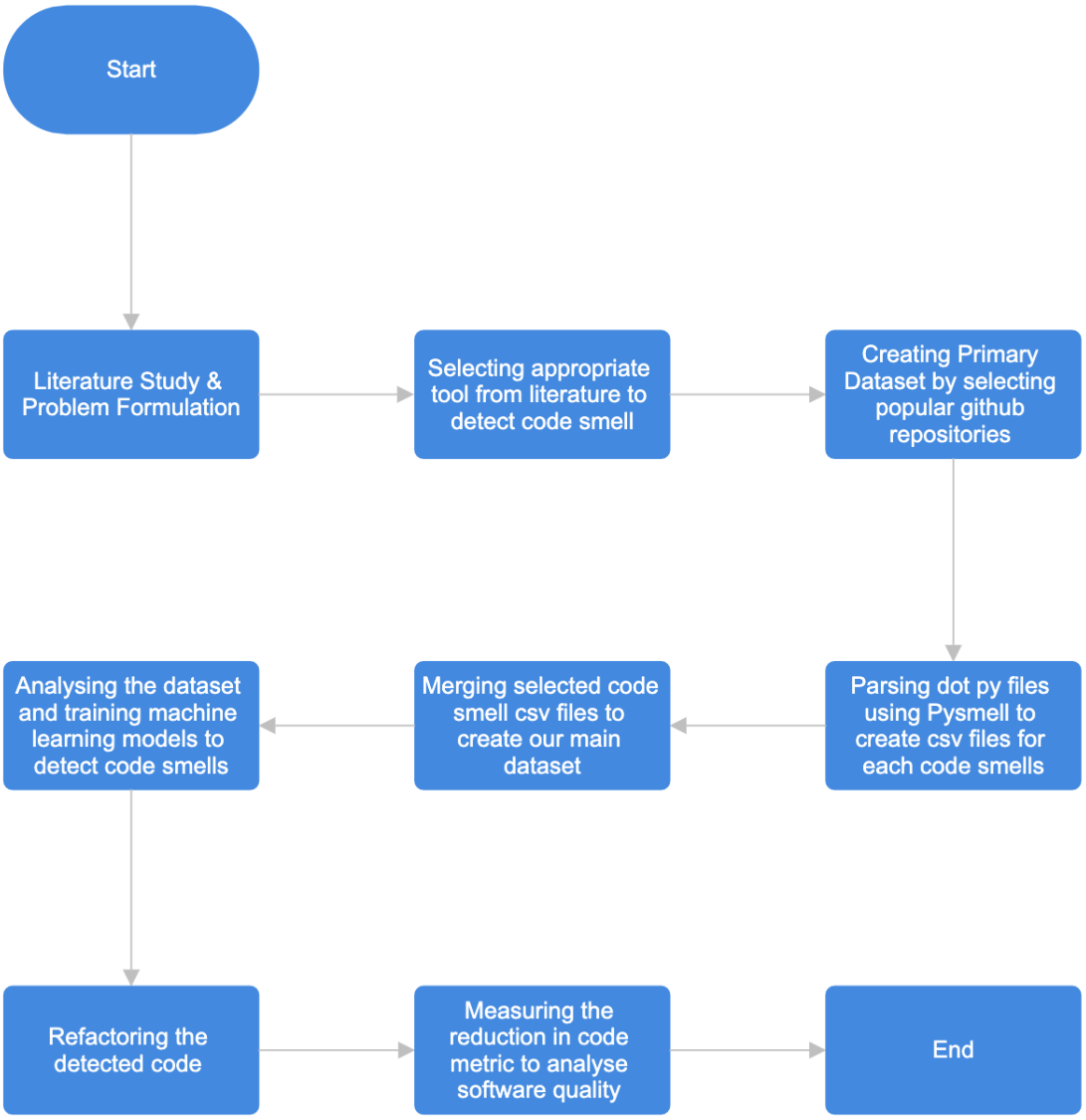


Figure 3.1: Flow Chart

Chapter 4

Dataset

4.1 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.

4.2 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 for an in-depth description of the tool. The architecture of Pysmell is outlined below:

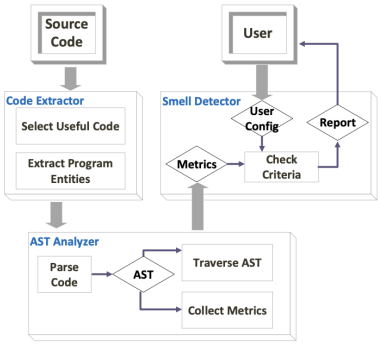


Figure 4.1: Architecture of Pysmell

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.

4.3 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.

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 5

Methodology

5.1 Code Smells and Metrics

Our work centers on the identification and mitigation of specific code smells, each posing unique challenges to software maintainability:

Large Class (LC): Characterized by an excessive number of lines of code within a class.
Long Method (LM): Denotes a method with an unwieldy length, impacting readability and maintainability.
Long Parameter List (LPL): Refers to a method or function burdened by an unusually high number of parameters.
Long Message Chain (LMCS): Occurs when multiple methods are invoked using the dot operator, leading to a lengthy chain of calls within an object.
Long Lambda Function (LLF): Identifies lambda functions with an excessive number of characters, deviating from their intended purpose as concise inline functions.

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 [4]. The table below 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 ≥ 35	CLOC: Class Line of Codes
Long Method (LM)	Function	MLOC ≥ 50	MLOC: Method Line of Codes
Long Message Chain (LMCS)	Expression	LMC ≥ 4	LMC: Length of Message Chain
Long Parameter List (LPL)	Function	PAR ≥ 5	PAR: Number of Parameters
Long Lambda Function (LLF)	Expression	NOC ≥ 70	NOC: Number of Character

Table 5.1: Metric-based Code Smells for Python

5.2 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 and the Sigmoid activation function for the output layer. 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 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.

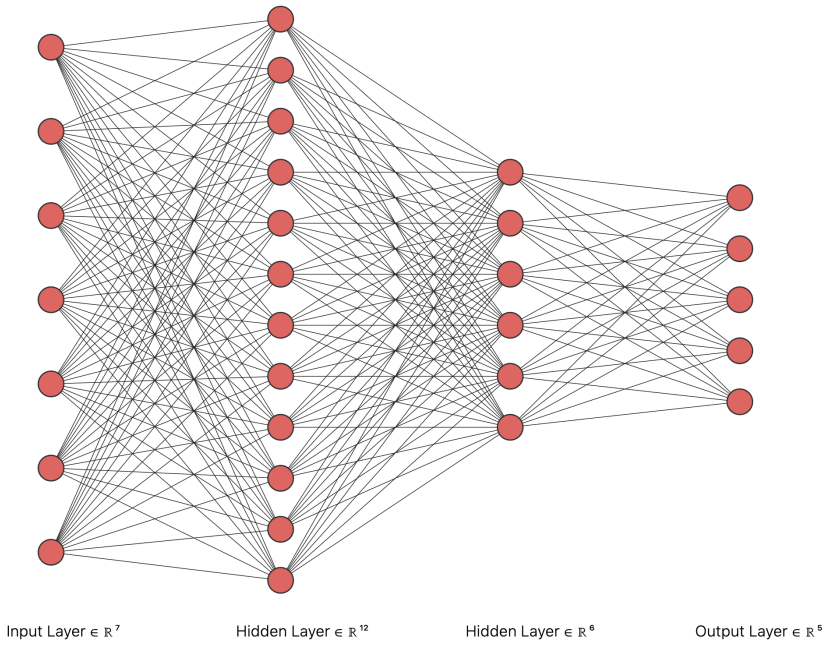
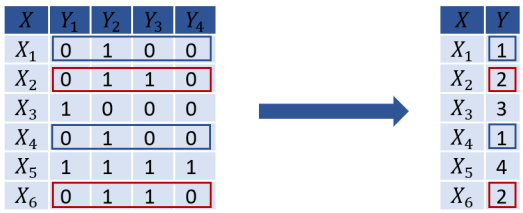


Figure 5.1: Architecture of The Neural Network

5.3 Ensemble Learning using Label Powerset

5.3.1 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. 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. The diagram below shows the classification procedure for the label powerset.



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 5.2: Label Powerset example

5.3.2 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 AdaBoost was the first and most used boosting technique for binary classification [9].

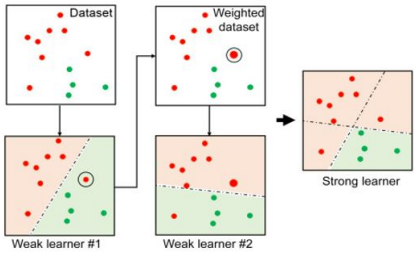


Figure 5.3: AdaBoost example

- XgBoost: XgBoost, which is short for extreme gradient boosting, is a popular ensemble technique that was developed by Tianqi Chen [9]. 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.

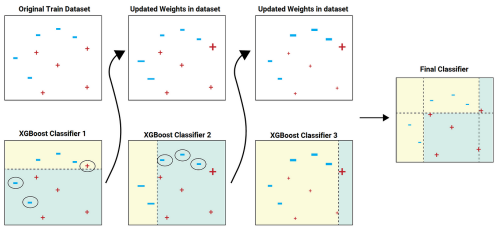


Figure 5.4: XgBoost example

- 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 [10].

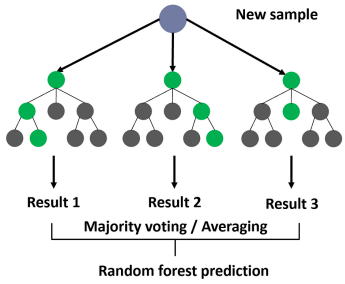


Figure 5.5: Random Forest example

5.3.3 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 5.2: Performance metrics for Ensemble Learning Models

From the above table 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 additon, XgBoost and Random Forest's hamming loss value was also very low compared to AdaBoost.

Chapter 6

Preliminary Analysis

6.1 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 chart, the x-axis represents the type of class and the y-axis represents the amount of code smell in each file. The figures are shown below:

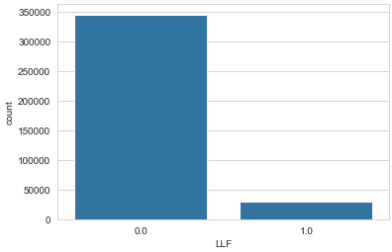


Figure 6.1: LLF

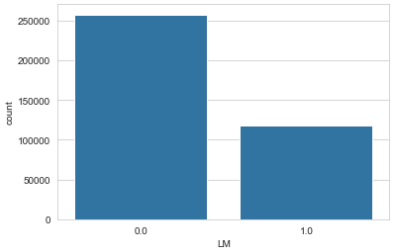


Figure 6.2: LM

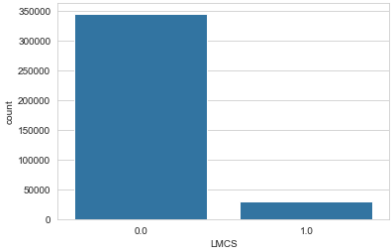


Figure 6.3: LMCS

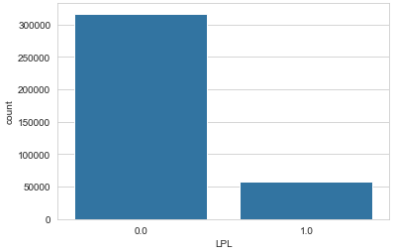


Figure 6.4: LPL

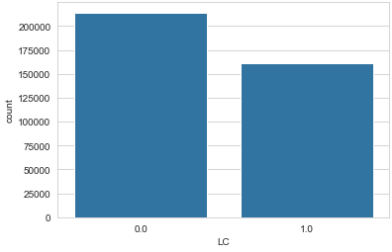


Figure 6.5: LC

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).

6.2 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 below:

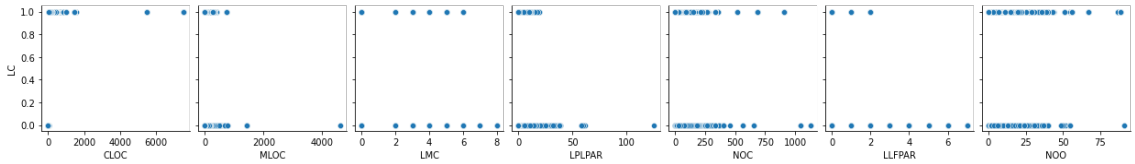


Figure 6.6

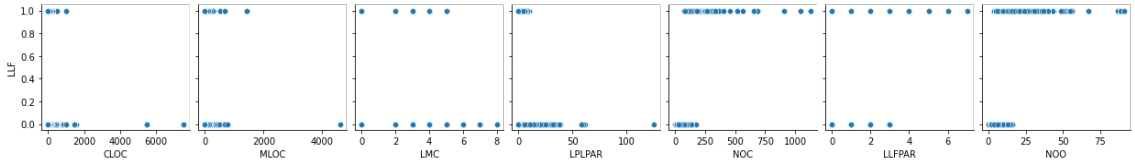


Figure 6.7

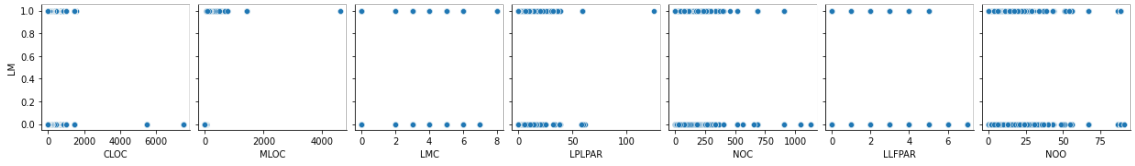


Figure 6.8

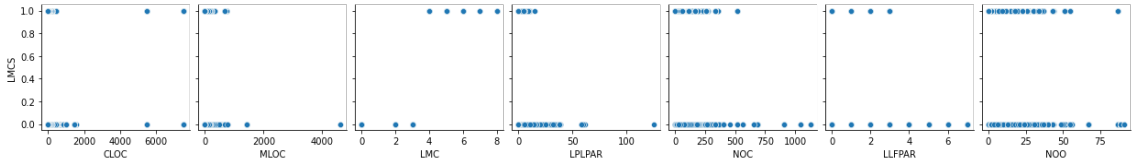


Figure 6.9

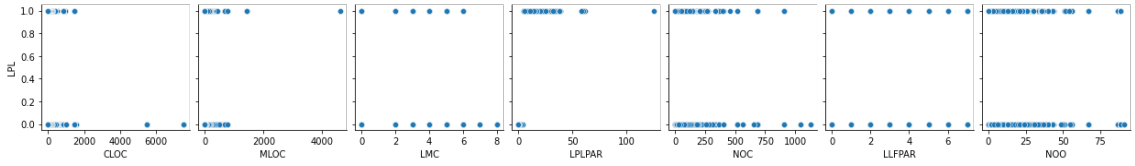


Figure 6.10

From the above scatter plot diagrams, 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.

6.3 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 below:

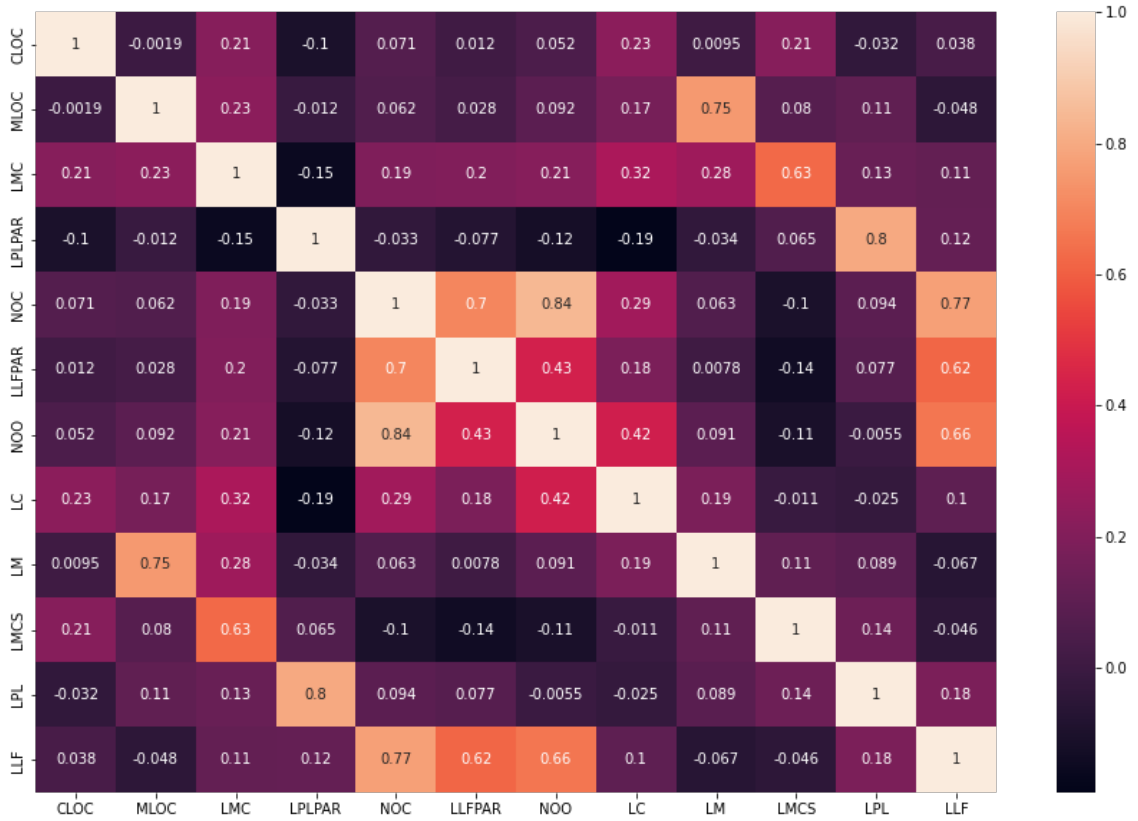


Figure 6.11

From the above heatmap, 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.

Chapter 7

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 [4] 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. Employing algorithms and model training, we aim to refactor identified code smells, subsequently measuring the resulting decrease in code smell and the corresponding enhancement in software quality. Our endeavor underscores the importance of preserving code cleanliness to ensure optimal maintainability in Python codebases.

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.