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

1st Jannatul Ferdoshi
Department of Computer Science and Engineering (CSE)
School of Data and Sciences (SDS)
Brac University
Dhaka, Bangladesh
jannatul.ferdoshi@g.bracu.ac.bd

2nd Kazi Zunayed Quader Knobo
Department of Computer Science(CS)
School of Data and Sciences (SDS)
Brac University
Dhaka, Bangladesh
zunayed.quader.knobo@g.bracu.ac.bd

3rd Shabab Abdullah
Department of Computer Science and Engineering (CSE)
School of Data and Sciences (SDS)
Brac University
Dhaka, Bangladesh
shabab.abdullah1@g.bracu.ac.bd

4th Mohammed Sharraf Uddin
Department of Computer Science(CS)
School of Data and Sciences (SDS)
Brac University
Dhaka, Bangladesh
mohammed.sharraf.uddin@g.bracu.ac.bd

5th Md. Aquib Azmain
Department of Computer Science and Engineering
School of Data and Sciences (SDS)
Brac University
Dhaka, Bangladesh
aquib.azmain@bracu.ac.bd

***Abstract*—Python has witnessed substantial growth, establishing itself as one of the world's most popular programming languages. Its versatile applications span various soft- ware 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 detec- tion and refactoring for Python code smells are nascent, our research contributes essential advancements.**

***Index Terms*—Python, Code Smells, Code Refactoring, Machine Learning, Code Analysis, Software Quality, Software Maintenance, GitHub Repos- itories iii**

I. Introduction

The success of any software depends heavily on maintaining code quality in the con- stantly 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 tech- nically 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 re- quired 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.

II. 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?

III. 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.

IV. Literature Review

A. Test Smell Detector for Python

The paper [1] begins by outlining the idea of code smells before gliding into the topic of test smells, which are essentially code smells resulting from subparly constructed test cases that have a negative effect on the production code. The writers, Goluben et al., figured that there was a substantial gap in the testing tool scenario by indicating that while test-smell detecting tools are hugely available for JAVA and Scala, they are noticeably absent for Python. The paper [1] talks about a novel tool called Pynose, which was created to fill out this gap. The first steps in their methodology included selecting specific test smells for analysis [1]. Goluben et al. was able to identify 33 different test smells in JAVA, Scala, and Android systems by performing a systematic mapping study of test smells. From the 33 identified smells 17 test smells were customized for Python's Unittest testing framework after performing a careful filtering process. The paper [1] also emphasizes how crucial it is for the framework of the tool to address Python-specific test smells.

Thereafter, the authors of the paper [1] started the creation of primary dataset, diligently gathering 450 project from GHTorrent, all meeting rigorous criteria: at least 10 contributors, no forks, 1000 commits, and a minimum criteria of 50 stars. The authors made use of a tool, PYTHONCHANGEMINER, that can track changes made to test files to spot Python-specific test smells within this dataset. Completing this step made the authors realize that there was a misuse of assert functions, leading to a specific test smell known as suboptimal assert. Moreover, Pynose's precision and accuracy in detecting test smells was evaluated using a secondary dataset, comprising of 239 projects.

Finally, Golubev et al. architected and built the Pynose tool in such a clever way so that it can be integrated as a plugin for PyCharm. Firstly, the tool parses and analyses the Python source code using JetBrains' IntelliJ Platform's Python Structure Interface (PSI), ensuring both syntactic and semantic examination [1]. After this, the tool selectively extracts classes belonging to unittest.TestCase. Lastly, the tool employs specific detector classes to detect test smells within the extracted classes. The tool intelligently presents the detected code smells in the integrated development environment (IDE) or saves them in a structured JSON format. Remarkably, the evaluation of the tool using the secondary dataset demonstrated an impressive precision rate of 94% and an equally commendable recall rate of 95.8%, reaffirming its efficacy in detecting and addressing test smells within Python codebases [1].

B. 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 [2]. 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 [2] 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 [2].

C. 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 [3] is to identify code smells and provide support for refactoring strategies, ultimately enhancing the software quality of Python programs.

At first, Chen et al, acknowledged that certain standard code smells may not be applicable in the context of Python, thus, they took on the task of classifying code smells relevant to Python. The authors of the paper [3] conducted vigorous review of online resources and reference manuals to find out the characteristics of Python code smells. Some of the prominent Python smells are: Large Class, Long Parameter, Long Method, Long Message Chain, and Long Scope Chaining [3].

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

At last, the authors built a tool, Pysmell, which is designed to detect code smells based on relevant metrics. The tool's architecture has three main components:
i) The code extractor, which cleverly extracts out the python files from a project that is required for further investigation.
ii) The Abstract Tree Analyzer constructs a abstract syntax tree from the extracted file and also collects the required metric that is set for a code smell.
iii) The Smell Detector, which detect code smells based on the metric that was found in the previous component.

To finish their study, they evaluated the Pysmell tool, achieving an impressive precision of around 98% and a mean recall of 100% in detecting code smells within Python systems. However, as the study was conducted purely on metric there are some biases on the results of the evaluation [3].

D. Code Smell Detection: Towards a Machine Learning-Based Approach

Francesca Arcelli Fontana, Marco Zanoni, and Alessandro Marino discussed the machine learning techniques, which are applied to identify code smells in software development in the paper [4]. Code smells are patterns of inefficiency that can seriously harm the quality and maintainability of software. To keep codebases readable and long-lasting in software engineering, it is essential to find code smells. This paper addresses the difficulties in detecting code smell and suggests a novel approach that makes use of machine learning to improve accuracy and efficiency in this procedure.

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

To get over these problems, the authors propose a code smell detection method based on machine learning. More precise and reliable detection strategies are needed, and they highlight the dearth of research on machine learning techniques in this area. The paper goes into great detail about the methodology, which consists of data collection, code smell selection, application of detection techniques, manual code smell candidate labeling, and machine learning classifier testing.

The authors have chosen and ranked a number of prominent and frequent code smells according to severity, including

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

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

Code smells are indicators of poor design choices that can negatively impact source code quality and maintainability. Code smell detection using machine learning (ML) approaches is investigated by Dario Di Nucci et al. [5]. The authors acknowledge the challenges in identifying code smell and discuss the potential advantages of using ML to these issues. They emphasize how programmers must choose between speed and excellent practices due to the increasing complexity of software systems. Code maintenance becomes more difficult when such tradeoffs result in code smells and other technical debt.

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

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

Dario Di Nucci et al. continue to question the generalizability of the results. They voice concerns regarding the potential impact of the dataset on the greater performance shown by Arcelli Fontana et al. The original work employed an imbalanced dataset, with instances impacted by a particular type of code smell and non-smelly cases included in each dataset for that particular sort of fragrance. In order to address these issues, the authors perform a repeatable analysis on a different dataset that includes code components affected by different code smells. The revised dataset more closely resembles real-world situations by less equally distributing stinky and non-smelly occurrences. The latest research claims that the dataset employed, not the innate capabilities of ML models, was the reason for the previous study's surprise success.

F. Code smells detection and visualization: A systematic literature review

Another paper by Craig Anslow [6], Jose Pereira dos Reis, Fernando Brito e Abreu, and Glauco de Figueiredo Carneiro, titled Code Smells Detection and Visualization: A Systematic Literature Review, conducts a systematic literature review (SLR) on the topic of code smell detection and visualization. This study examines the various tools and approaches used to detect software code smells as well as the extent to which these approaches have benefited from visualization. Code smells, also referred to as bad odors, are problems with software design and code that lower program quality and make maintenance more difficult. The lifespan and general well-being of software systems depend heavily on the ability to recognize and address code smells.

The results of the SLR reveal several key findings:

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

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

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

Zhifei Chen and colleagues delve into code issues related to Python. They point out that Pythons' simple structure and dynamic characteristics have led to a focus on analyzing code issues compared to rigid languages, like Java and C sharp [7]. The main goal of their study is to identify and categorize code issues in Python programs while also examining how they affect software maintainability. The paper outlines ten code problems in Python. Explains a method, for detecting them using different threshold setting approaches; machine learning, experience-based, and statistical analysis.

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

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

H. Comparing and experimenting machine learning techniques for code smell detection

Francesca Arcelli Fontana et al. presented research on using machine learning (ML) approaches to detect code smells [8]. For many years, code smells have been a major issue in software quality improvement. Code smells are difficult to detect since various individuals have diverse ideas and solutions, and these challenges do not follow any common principles of solution. So these issues can be resolved if we can identify the code smells. This research investigates how machine learning (ML) methods may be used to identify code smells automatically.

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

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

I. Python code smells detection using conventional machine learning models

Rana S. and Hamoud A. [9] 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. [9] have created their datasets containing Python code smells. This is done because Python is the most used language in creating data science and machine learning models. Therefore, they have designed datasets based on two criteria. Firstly, they have extracted code smells based on class level and method level. Secondly, they have extracted those code smells that are most present in the Java programming language. As a result, they have chosen Long Methods and Large Class code smells for their Python datasets. In the creation of the Python code smell dataset, four Python libraries—Numpy, Django, Matplotlib, and Scipy have been used by the researchers. The resultant dataset consisted of 18 different sets of features, which were categorized as smelly and non-smelly for each code smell. The datasets were further validated through the use of verified tools like Radon, which ensured the quality of code metrics. To ensure machine learning models' high performance in detection, two data pre-processing steps—feature scaling and feature selection—were carried out on the datasets. Later, to detect code smells in the source code, six machine learning algorithms like support vector machines (SVM), random forest (RF), stochastic gradient descent (SGD), decision trees (DT), multi-layer perceptron (MLP), and logistic regression (LR) were applied. Furthermore, to assess the performance of the machine learning models, two different performance calculation measurements were taken into consideration: the Matthews correlation coefficient (MCC) and accuracy.

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

J. Code Smell Detection Using Ensemble Machine Learning Algorithms

Seema et al. [10] 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. [10] 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. [10] 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. [10] 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.

V. Workplan

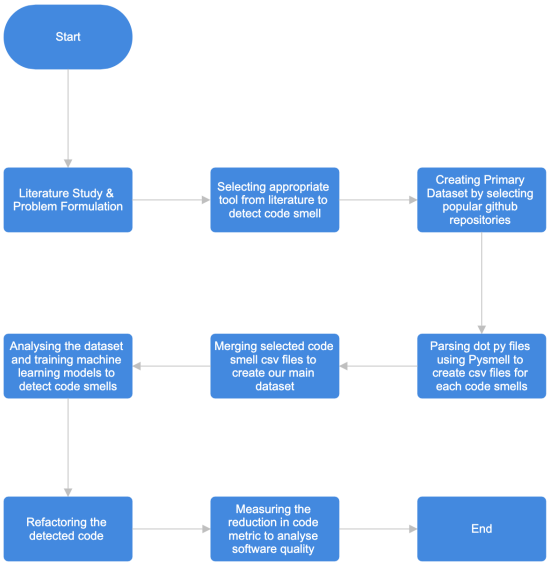

Fig. 1. Flow Chart

VI. Dataset

A. 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.

B. 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 in figure 2.

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

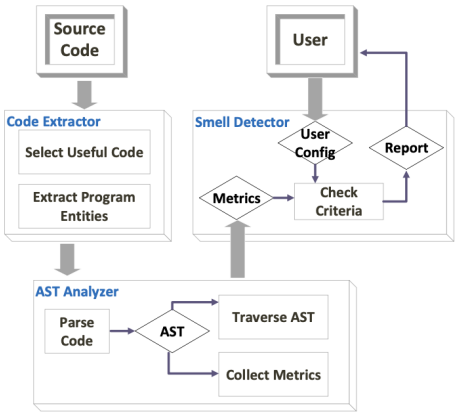

Fig. 2. Architecture of Pysmell

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 of the code smell discussed above, the Pysmell tool produced a comma-seperated file. These comma-seperated files were crucial as it gave details about each code smell 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 details helped us to understand the code smells better and visualize them in the python file that the code smell occurred , enabling us to apply a thorough analysis of our second dataset.

C. Tertiary Dataset

We created our final dataset by selecting five code smells: Large Class (LC), Long Method (LM), Long Message Chain (LMCS), Long Parameter List (LPL), and Long Lambda Function (LLF). In the secondary dataset, each of code smell had its own comma-separated file. However, to make our work flow smoother, we cleverly merged all these code smells into a single comprehensive comma-separated file.

While merging, we made a noteworthy observation: the occurrence of each code smell per project was relatively low, but enough for it to be considered a significant issue in the software development field. This made us realize that the dataset was hugely imbalanced, so we took proactive steps to address this problem. Specifically, we tried making our dataset balanced by carefully selecting an equal number of rows representing both smelly and non-smelly rows for each code smell.

The final dataset was named the code smell dataset. This dataset is a multi-labeled dataset, which underwent exploratory data analysis, data preprocessing, model training, and model evaluation.

VII. Methodology

A. Code Smells and Metrics

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

Large Class (LC): A class that contains excessive number of lines
Long Method (LM): A method that is huge in length making the code difficult to maintain.
Long Parameter List (LPL): A method that takes in a lot of parameters as argument.
Long Message Chain (LMCS): Occurs when multiple methods are called using dot. This leads to lengthy calling of methods.
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 [3]. Table I illustrates the code smell, at which level it occurs, the metric, and the threshold of the metric at which the code smell occurs.

B. 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 shown in figure 3 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.

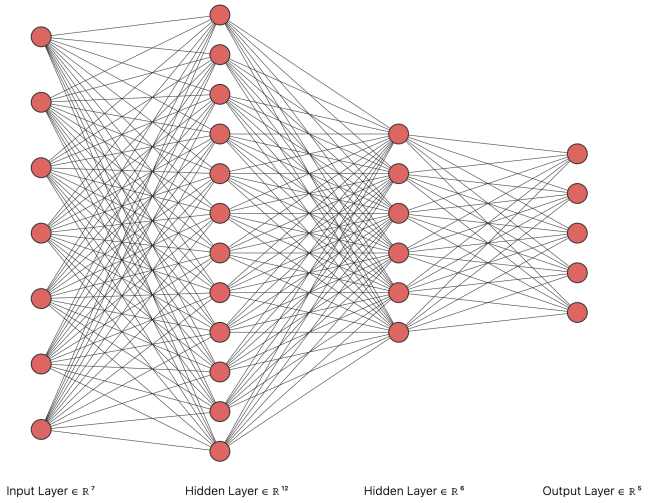

Fig. 3. Architecture of Neural Network

C. Ensemble Learning using Label Powerset

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. Figure 4 shows the classification procedure for the label powerset.

TABLE I
METRIC-BASED CODE SMELLS FOR PYTHON

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

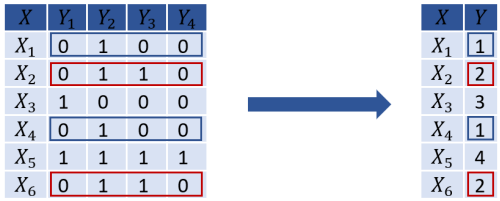

Fig. 4. Label Powerset example

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, shown in figure 5, 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].

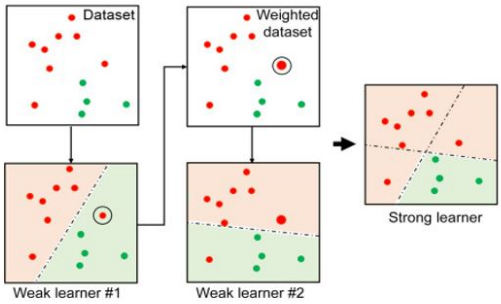

Fig. 5. AdaBoost example

- XgBoost: XgBoost, shown in figure 6, 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.
- Random Forest: Random Forest, shown in figure 7, 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].

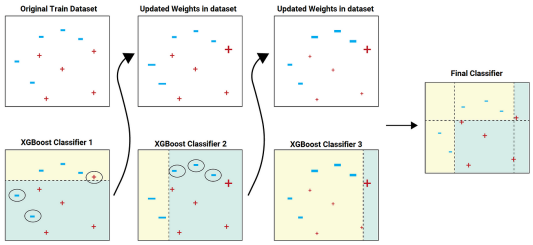

Fig. 6. XgBoost example

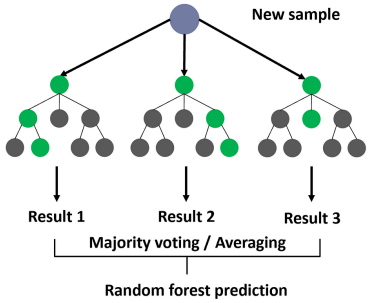

Fig. 7. Random Forest example

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.

TABLE II
PERFORMANCE METRICS FOR ENSEMBLE LEARNING MODELS

Ensemble Models	Accuracy	Precision	F1-Score	Recall	Hamming Loss
AdaBoost	69%	96%	83%	73%	0.06
XgBoost	100%	100%	100%	100%	7.98×10^-6
Random Forest	100%	100%	100%	100%	5.33×10^-6

From Table II 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.

VIII. PRELIMINARY ANALYSIS

A. 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 in the next page.

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).

B. 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 given in the next page.

From the 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 the scatter plot of Large Class, 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.

C. Correlation of the code smells and their metrics

A statistical measure called correlation indicates how much two variables change together. A common way to represent correlation is with the Pearson correlation coefficient, which has a range of -1 to +1. A positive correlation means that if the value of one variable increases, then the value of the other variable also increases (value $>$ 0). A negative correlation means that if the value of one variable increases, then the value of the other variable decreases (value $<$ 0). If there is no linear relationship between the variables, then there is a correlation of 0 (value = 0). Therefore, we have created a heatmap to find out the correlation between the input features and class labels using the seaborn library. The heatmap is shown in figure 10.

From the 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.

IX. 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 [3] 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.

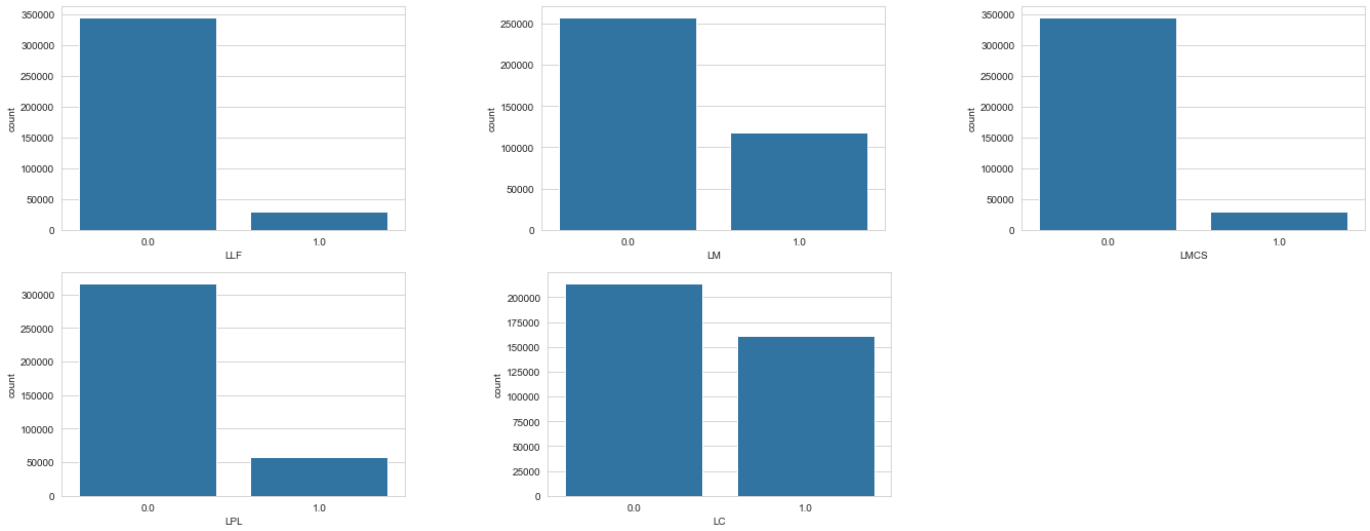

Fig. 8. Class distribution of LLF, LM, LMCS, LPL & LC

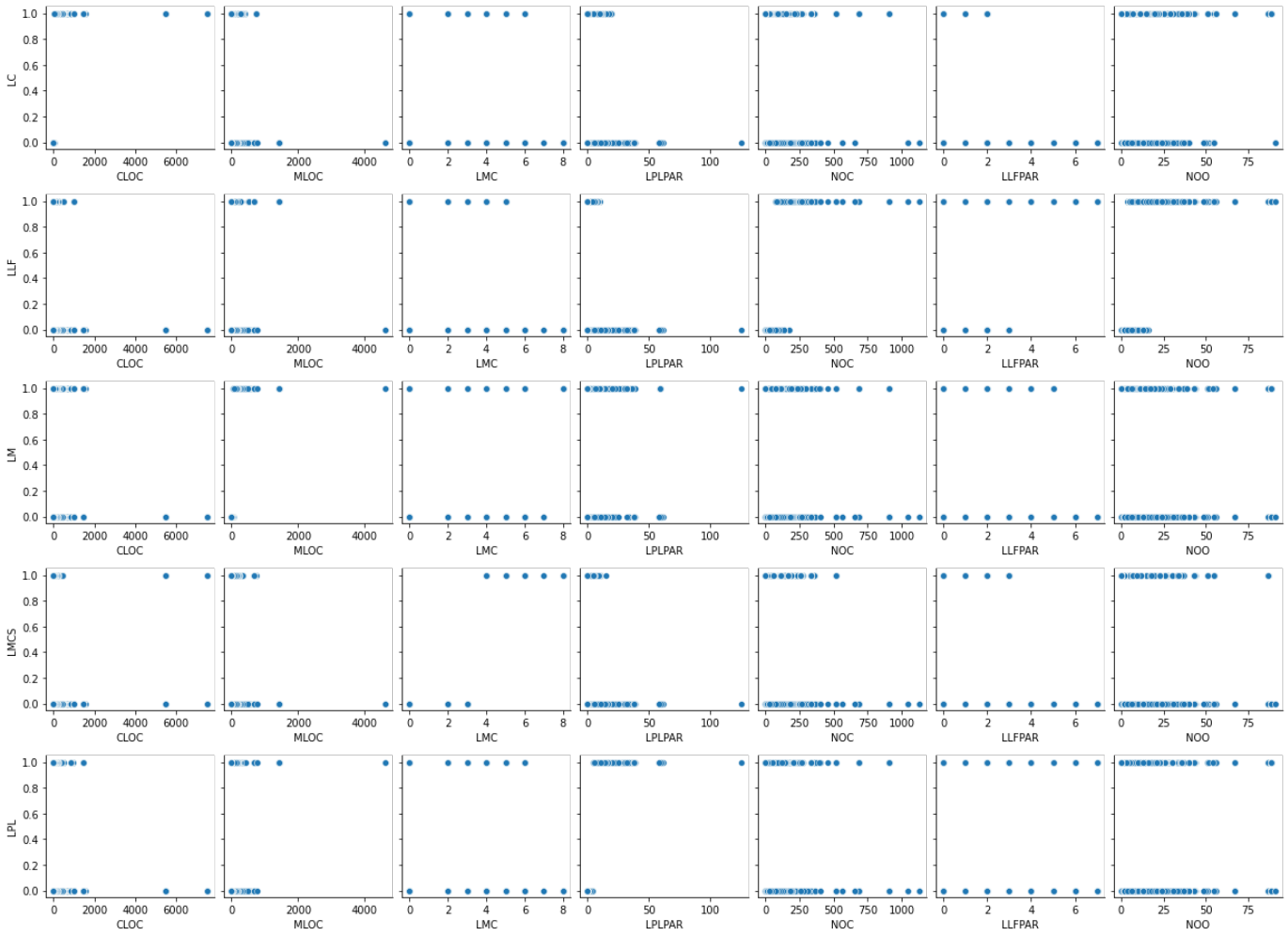

Fig. 9. Scatter Plot

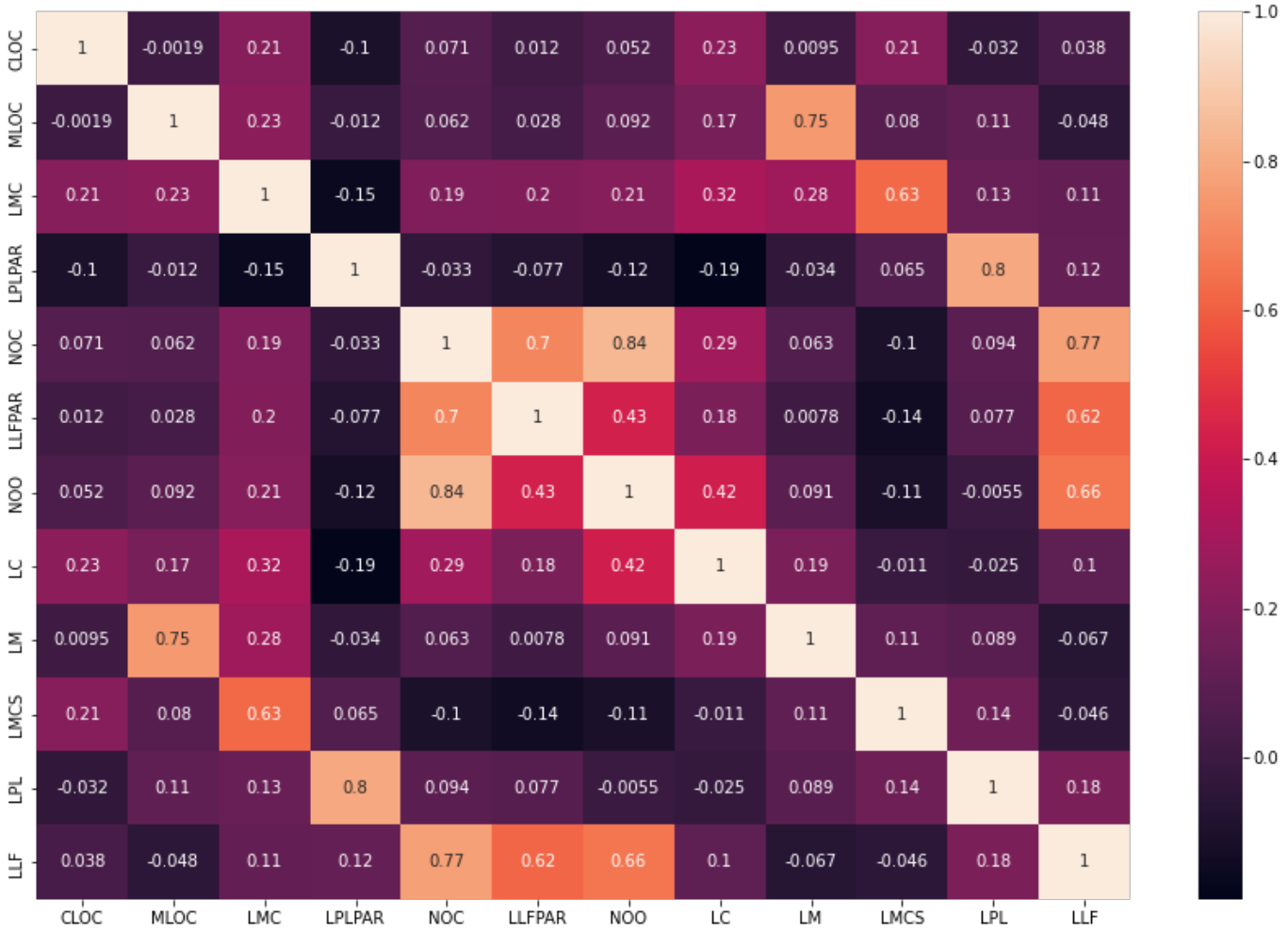

Fig. 10. Correlation Heatmap of Code smells and their correspoding Metrics

REFERENCES

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] J. Reis, F. Brito e Abreu, G. Carneiro, and C. Anslow, "Code smells detection and visualization: A systematic literature review," 12 2020.

[7] 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. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584916301690

[8] 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, 06 2015.

[9] R. Sandouka and H. Aljamaan, "Python code smells detection using conventional machine learning models," *Empirical Software Engineering*, 05 2023.

[10] S. Dewangaan, R. S. Rao, A. Mishra, and M. Gupta, "Code smell detection using ensemble machine learning algorithms," 10 2022.