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

MOHAMMED SHARRAF UDDIN, BRAC University, Bangladesh
KAZI ZUNAYED QUADER KNOBO, BRAC University, Bangladesh
JANNATUL FERDOSHI, BRAC University, Bangladesh
SHABAB ABDULLAH, BRAC University, Bangladesh
MD. AQUIB AZMAIN, BRAC University, Bangladesh

Python has witnessed substantial growth, establishing itself as one of the world's most popular programming languages. Its versatile applications span various software and data science projects, empowered by features like classes, method chaining, lambda functions, and list comprehension. However, this flexibility introduces the risk of code smells, diminishing software quality, and complicating maintenance. While extensive research addresses code smells in Java, the Python landscape lacks comprehensive automated solutions. Our paper fills this gap in two ways. Firstly, by constructing a dataset using a tool from existing literature, Pysmell [5]. The tool, given a project directory, determines python files and produces comma sepa- rated files for code smells that are present in the python file. We create a dataset containing github projects and run the tool on our dataset. Then we select five comma separated code smell files: Large Class, Long Method, Long Lambda Function, Long Parameter List and Long Message Chain. The comma separated files are then combined to produce a multi-label dataset of code smells. Ensemble techniques and neural networks are trained on the dataset to analyse the performance of machine learning models in predicting code smells given a metric. Secondly, our approach extends to designing and building a simple automated refactoring algorithm, aiming to reduce long method code smells by extracting out large if-else statements and elevate overall software quality. In a landscape where automated detection and refactoring for Python code smells are nascent, our research contributes essential advancements.

CCS Concepts: • **Software and its engineering → Software maintenance tools**;

Additional Key Words and Phrases: Python, Code Smells, Code Refactoring, Ma- chine Learning, Code Analysis, Software Quality, Software Main- tenance, GitHub Repositories

ACM Reference Format:
Mohammed Sharraf Uddin, Kazi Zunayed Quader Knobo, Jannatul Ferdoshi, Shabab Abdullah, and Md. Aquib Azmain. 2025. Enhancing Software Quality: Python Code Smell Detection using Machine Learning techniques and Refactoring Long Methods using Extract Method Algorithm. *J. ACM* , (January 2025), 17 pages.
https://doi.org/XXXXXXX.XXXXXXX

Authors' addresses: Mohammed Sharraf Uddin, BRAC University, Bangladesh; Kazi Zunayed Quader Knobo, BRAC University, Bangladesh; Jannatul Ferdoshi, BRAC University, Bangladesh; Shabab Abdullah, BRAC University, Bangladesh; Md. Aquib Azmain, BRAC University, Dhaka, Bangladesh.

ACM 0004-5411/2025/1-ART
https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

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[3]. These problems show up as unnecessary code, excessively complicated structures, or poor design decisions. A code smell is a cue for developers to dig deeper into their code, much as a bad smell might indicate underlying issues. It's important to identify and fix code smells for a number of reasons. First of all, it has an immediate effect on the development process's efficacy and efficiency. Code bases with a lot of code smells are more challenging to comprehend, alter, and extend. As a result, it is more likely that while making modifications, genuine problems will be introduced or unwanted side effects will be produced. Additionally, code smell can raise maintenance costs and reduce the agility needed in the quick-paced software development settings of today. Technical debt, which builds up as a result of unchecked code stench, may hinder creativity and impede the advancement of software. It is evident that code smells can hinder the quality of a software and dealing with such code smells can be frustrating for developers. After researching on automated detection and refactoring of software, we found that many tools have been build to address this issue for JAVA programming language. Nevertheless, we wanted to take an approach that is more future proof by introducing machine learning and deep learning to detect code and refactor code smells in python. Our paper makes the following contributions:

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

The rest of this paper is organized as follows: In Section 2 we have discussed the problem statement of our research, in Section 3 we have briefly described the objective of our research, in Section 4 we reviewed some related works, in Section 5 we have built our own Dataset, in Section 6 we have given a complete and elaborated methodology of our paper, in Section 7 we have done analysis of our smell detector and our refactoring algorithm, in Section 8 we have shown the limitations of our work and lastly in Section 9 we concluded our paper.

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?

3 RESEARCH OBJECTIVE

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

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

Furthermore, the paper will discuss how a refactoring algorithm was implemented to refactor one of the most prominent code smell, long method, without human intervention, improving the quality of the code base. We will then evaluate the effectiveness and efficiency of the proposed algorithm by running it on four different python files and finding out the limitations.

4 LITERATURE REVIEW

The field of code smell detection has greatly advanced with the help of machine learning (ML). Researchers have created tools that are designed for specific programming languages like Python and Java. These tools help find and fix code problems that can make software harder to maintain and improve.

Firstly, there are already many tools for detecting code smells in Python. An author named Golubev et al. developed Pynose, a tool for finding test smells that are also called code smells in Python's Unittest framework, the result was very accurate in the tests[21]. Another author Chen et al. worked on Python-specific code smells with a tool that is called Pysmell. Pysmell was very accurate because it used metrics to detect code smells[5]. Additionally, Rana S. and Hamoud A. used standard ML models to find often-missed code smells in Python, achieving impressive accuracy[18]. At the same time, there have been important developments in tools for Java. Nagy et al. created FaultBuster, which helps developers refactor code continuously and has been effective in large codebases[24]. Seema et al. looked into how different metrics affect the tool's performance[9] and used both machine learning and deep learning to find different types of code smells very accurately.

Machine learning has added new capabilities in this area. Arcelli Fontana et al. proposed the idea of training ML models to detect code smells using a large number of features from the source code in[11][2], motivating future work. Di Nucci et al. explored the automated identification of code smells using ML and noticed that different datasets can greatly impact how well this tool works[10]. Systematic reviews have provided valuable insights into popular techniques for detecting code smells and highlighted the need for improved visualization tools to manage these issues effectively[17]. Also, comparative work[6] increased for Python metric-based code smells we get to know how different choices affect software maintainability. Currently, innovative methods like

the tuck transformation as proposed by Lakhotia and Deprez[14] or techniques like those from the AutoMeD tool to extract fragments from long methods[22] in systems are demonstrating new ways how software can be organized better which also could lead to more opportunity of analysis.

The results of these efforts can serve as a solid foundation for further research in the field of code smell detection. They demonstrate the significance of machine learning and automated techniques in terms of enhancing the quality of software and assisting developers in working more effectively.

5 DATASET

5.1 Primary Dataset

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

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

5.2 Secondary Dataset

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

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

5.3 Tertiary Dataset

In the culmination of our research efforts, we curated our final dataset by focusing on five distinct code smells: Large Class (LC), Long Method (LM), Long Message Chain (LMCS), Long Parameter List (LPL), and Long Lambda Function (LLF). Each code smell was initially stored in its own dedicated comma-separated file. To streamline our analysis, we strategically merged all these code smells into a single comprehensive comma-separated file. This resulted in the creation of a dataset where each row can be associated with one or more class. This area of problem is known as multi-label classification [8]

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 pre-processing, model training, and model evaluation.

6 METHODOLOGY

6.1 Code Smells and Metrics

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

Large Class (LC): A class that contains excessive number of lines [5].
Long Method (LM): A method that is huge in length making the code difficult to maintain [5].
Long Parameter List (LPL): A method that takes in a lot of parameters as argument [5].
Long Message Chain (LMCS): Occurs when multiple methods are called using dot. This leads to lengthy calling of methods [5].
Long Lambda Function (LLF): Identifies lambda functions with an excessive number of characters, deviating from their intended purpose as concise inline functions [5].

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 [5]. 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.

Table 1. 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

6.2 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, as it is produces less computation due to some neurons being inactive [20], and the Sigmoid activation function for the output layer, as this is classification problem sigmoid functions help in transforming the output to probability space thus performing better than other activation function [20]. 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 1, 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.

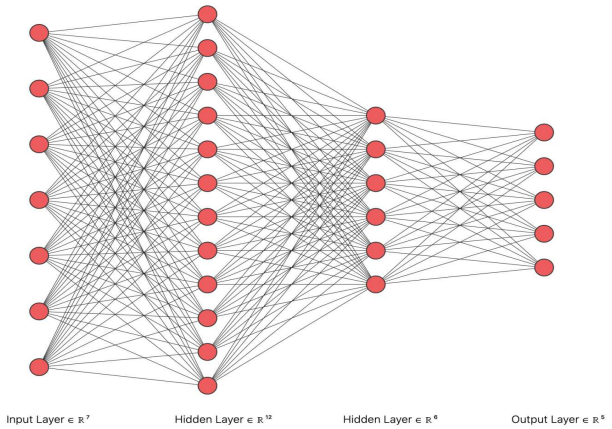

Fig. 1. Architecture of The Neural Network

6.3 Ensemble Learning using Label Powerset

6.3.1 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 [16]. 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 2 below shows the classification procedure for the label powerset.

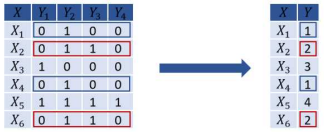

Fig. 2. Label Powerset example

6.3.2 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 for classifying binary kinds of problems, AdaBoost won the race. In addition, it is the most familiar boosting approach for such two-fold classification problems [9].

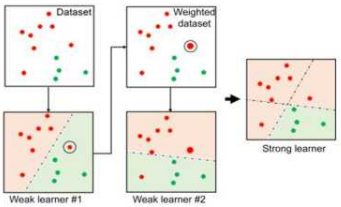

Fig. 3. AdaBoost example

- 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.

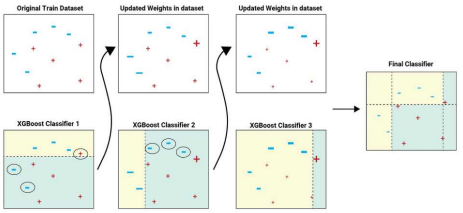

Fig. 4. XgBoost example

- 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, shown in figure 5, in order to enhance accuracy and lower over-fitting [18].

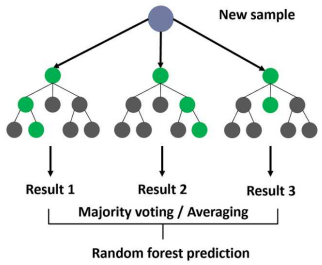

Fig. 5. Random Forest example

6.3.3 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 2. Performance metrics for Ensemble Learning Models

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.

6.4 Refactoring

6.4.1 Refactoring of Long Method. As RQ1 and RQ2 which have already been addressed, RQ3 addresses the fact of incorporating an intelligent algorithm to automate the process of refactoring code smell. According to our analysis, Large Class and Long Method were the most popular contributing bad smells in python files. Hence, we have chosen Long Method as our prime code smell to restructure it. Almost every author in their papers has mentioned Martin Fowler and Kent Beck [1, 13, 15, 19, 23] who brought enlightenment in the software code smells and refactoring

process. Martin and Kent [12] vividly describes the art of refactoring. It is a process of creating a change in a software structure without changing the external composition of the code, however bringing an enhancement in the internal composition of code. Refactoring aids in eradicating code smells, improving the software quality of the system and diminishes the cost of maintaining a code base. Therefore, reducing the possibility of introducing new software bugs and also identifying bugs if there are any. It is important to understand that refactoring is not synonymous to rewriting a code script as refactoring fails to alter the functionality of the code script [1]. Refactoring a python long method code smell manually can have several drawbacks like immense time consumption, tedious and prone to making errors while refactoring. As a result an automated process of refactoring will bring ease in developers' minds.

6.4.2 Long Method and its refactoring solution. Long method is one of the most rampant code smells in almost every software project [19]. As the name suggests, it deals with methods (functions) in programs and it falls under the category of 'bloaters'. Bloaters are referred to code, methods and classes that it's length have increased to such an enormous extent that it becomes extremely complex to work with. Long method gives developers a hard time in reading and understanding the code scripts as the method consists of multiple conditions, loops, variable declarations and data operations. The problem of long methods can be decoded by decreasing the complexity of the method. This is done through pointing out those parts of the function that need explanation and separate it into a new method [1].

Several techniques are established by Martin and Kent [12] to tackle long methods and (Agnihotri & Chug, 2020) have spotted the most familiar refactoring approaches like move method, extract class and extract method. In our study, we have used the principles of the extract method in order to build the architecture of our automated refactoring tool. Extract method is the process where part of the method can be divided and a new method is created for that part. Codes that need explanation are extracted and pasted in the new method. In addition, the necessary aspects like variables and parameters of the extracted method are exported as new parameters to the newly built method. (Fowler & Kent, 2018) have explained the benefits of extracting a method. Those benefits are that the extract method gives rise to the probability of other methods to use a method where it is properly refactored and improves the readability of the higher-level methods.

6.4.3 Detailed Architecture of the Proposed Refactoring Algorithm. In this segment, we will be presenting and highlighting every detail of our proposed model that aims to mechanize the entire procedure of refactoring a Long Method. Taking conceptual ideas of extracting a method from (Fowler & Kent, 2018) book and the following related research papers [14, 22] on refactoring Long Methods of 'Java' programs. We have formulated our proposed approach on refactoring python Long methods.

The flow chart presented in Figure 6 is the high-level design of our suggested algorithm. In the first step, a python file containing long methods is given as an input to our main function which then carries out four important mechanisms. In the second step, the structure of the input file is checked like the indentation and spacing of the file. If there is any indentation error then it is corrected before moving to the next course of action. Moving on to the third and fourth step, long methods are thoroughly examined and conditional statements are found out and removed from the long method and placed in the new method. Lastly, the input file is refactored where complex if-else statements are converted into new methods (basic statements) and are called in the input file. Thus, the system reduces the overall line numbers of the input long method. The following sections give a comprehensive explanation of each step of the suggested algorithm.

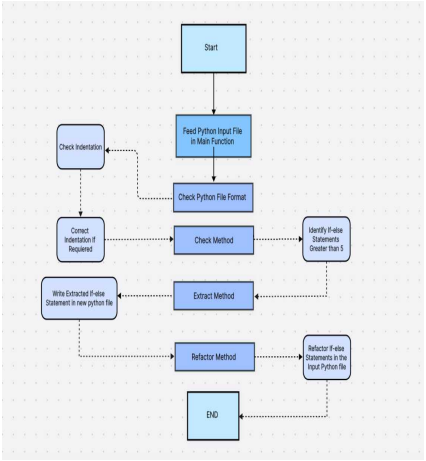

Fig. 6. Flow chart of the naive refactoring algorithm

A. Exemplifying the Input File

To illustrate the mechanism of the algorithm we have created a simple program named classify_person, which consists of a long method of our own. This python long method is used to classify a person based on age, gender and income. The size of the method is 67 lines which exceeds the threshold of MLOC as discussed in Table I. For better apprehension and automation of the refactoring process we have only considered conditional (*if-else*) statements that need to be extracted. As a result, the newly created method contains a gargantuan if-else statement making the method complex to refactor.

B. Composition of the Main File

This classify_person function is fed as an input to our main program. The main function then carries out and orchestrates sequentially the four key services in order to refactor the example long function classify_person. The four important branches of the main function include check_file_format.py, check_method.py, extract_method.py & refactor.py. However, if there is no large conditional statement in the input method then the main function will generate an output telling that there is no presence of conditional statement for extraction and refactoring. Figure 7 and the code snippet in listing 1 displays the functionality of the main function:

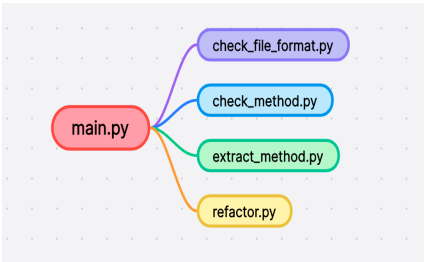

Fig. 7. Structure of the main.py

```
from check_method import check_method
from extract_method import extract_method
from refactor import refactor
from check_file_format import check_file_format

def main(filename):
    # Your main function to orchestrate everything
    check_file_format(filename)
    data, long_condition = check_method(filename)
    print(long_condition)
    if len(long_condition) != 0:
        functions = extract_method(data, long_condition)
        refactor(functions, long_condition, filename)
    else:
        print("There is no if statement that is between 6 and 50")


if __name__ == "__main__":
    main('Example Codes/example_code.py')
```

Listing 1. main.py

C. Inspecting the input python file

In this section, the example_code.py is investigated exhaustively to find any unevenly indented code. This is done in a series of steps. At first the example_code.py is read line by line and we have used 'rb' (read binary) in order to not exclude any white spaces in the input file. Secondly, since the file is formatted to binary we have used 'utf-8' decoder to decode each binary line of code back into string data type. Moving on, each line is then added in an array data structure and the array is passed in a loop for checking and fixing indentation error. Lastly, the modified strings of lines are appended in another array called new_lines and it is further written in the existing input python file.

D. Analyzing the Long Method

The actions in the check method are segmented into three parts. The first and foremost task of the check method is to identify and keep count of the number of if-else statements. The criteria for being a long conditional statement is that if the size (*number of lines*) of the conditional statement is greater than 5, then the check method will regard it as a long conditional statement. All the local and global variables that are associated with the conditional statements need to be pointed out and passed as a parameter for the new function created. Additionally, the starting and ending line number of the if-else statement need to be accounted for so that the new function will be aware from which line number to call the new function. To keep track of the number of if-else statements and variables, they are placed in a dictionary named long_condition. The key of the dictionary is the number of conditional statements while the values are the start and end of the conditional statement and variables. The figure 8 distinctly depicts the output of the long_condition from example_code.py. According to the example_code.py, there are two long if-else statements that are greater than 5, as a result there are two items in the dictionary. The first conditional structure had 4 parameters while the second condition had 3 parameters. Another important dictionary is also created in the check_method which is quite crucial in the extract_method section. This dictionary

consists of the line number as key of the python dictionary and code on that particular line number as value for the python dictionary. Both of these dictionaries are passed as parameters of the extract method.

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

Fig. 8. Output of the dictionary long_condition for the input exmaple

E. Configuration of the extract method

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

F. Refactoring the input long_method

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

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

Listing 2. example_code.py after being refactored

From the above listing 2 it can be observed that our unsophisticated algorithm is successful in shortening the Long Method (*classify_person*) of the input file example_code.py. Since two conditions were present in the input file, two methods have been created, imported and called in the appropriate location and other lines of code have remained untouched. The number of lines has been drastically reduced from 67 to 6 lines which is significantly below the benchmark set for Long Method ($MLOC \geq 50$).

7 RESULT ANALYSIS

7.1 Analysis of Code Smell Detection

7.1.1 Data Analysis. To understand the relationship between the input features and the output class label, we have analyzed the data. This inspection of data was graphically inspected with the aid of Python libraries like 'seaborn' and 'matplotlib'. At first, we plotted the number of code smells for each type of code smell in a bar chart to estimate the number of code smells that were present in each py file of the projects. In the bar charts represented in figure 9, the x-axis represents the type of class and the y-axis represents the amount of code smell in each file.

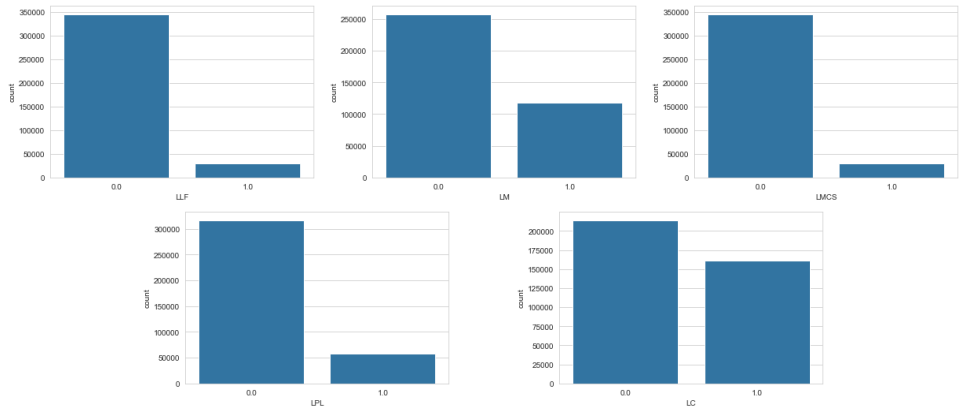

Fig. 9. Class distribution of LLF, LM, LMCS, LPL & LC

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

7.1.2 Relationship between input features and the class label. To examine any relation between all the code smell metrics (input features) and the final class label, like the presence of Python code smell, we have plotted a scatter plot diagram. The scatter plot diagram gave us insightful information about their relationship. This was done through the use of the pairplot function of the seaborn library. The scatter plot diagrams for each code smell are shown in figure 10, 11, 12, 13 and 14.

From figures 10 to 14, 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 10 above, we can see that most of the Large Class (LC) has occurred due to its own metric CLOC, but it is also evident that there are some of the Large Class smells that happened due to other metrics like MLOC, LMC, as well as the rest of the metrics.

7.1.3 Correlation of the code smells and their metrics. A statistical measure called correlation indicates how much two variables change together. A common way to represent correlation is with the Pearson correlation coefficient, which has a range of -1 to +1. A positive correlation means that if the value of one variable increases, then the value of the other variable also increases (value > 0). A negative correlation means that if the value of one variable increases, then the value of the other variable decreases (value < 0). If there is no linear relationship between the variables, then there is

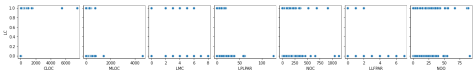

Fig. 10. Relation between Large Class and other Code Smells

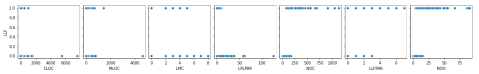

Fig. 11. Relation between Long Lambda Function and other Code Smells

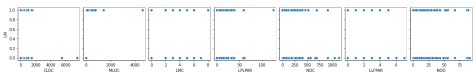

Fig. 12. Relation between Long Method and other Code Smells

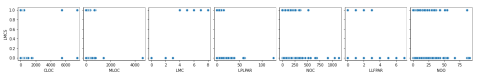

Fig. 13. Relation between Long Message Chain and other Code Smells

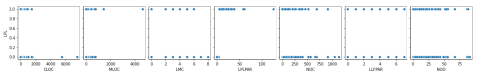

Fig. 14. Relation between Long Parameter List and other Code Smells

a correlation of 0 (value = 0). Therefore, we have created a heat-map to find out the correlation between the input features and class labels using the seaborn library. The heat-map is shown in figure 15.

Fig. 15. Correlation Heat-map

From figure 15, we can see that there is a strong positive correlation between the code smell and their code smell metrics. Besides, there is a weak correlation between code smells and the code smell metrics that are not of their type.

7.2 Analysis of the Refactoring Algorithm

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

7.2.1 File 1: Basic Example with a a long if-else statement. The example code.py consisted of two simple and long if-else statements. The first statement was inside a for-loop and the second statement was on its own. The for-loop was added to check if our algorithm could handle different

levels of indentations and some comments were also added to check if our algorithm could ignore these comments. After running our algorithm on the file we observed that even though "num" is a local variable, which was declared inside the if-else statement, it got passed as a parameter to method 1. Moreover, the new methods that were created return a value thus it should have been stored in a variable rather than just calling it. Hence, some manual adjustment is needed to make the code work properly. Even though our algorithm did not work, it correctly identified the part of code that needed refactoring and almost provided a neat solution.

7.2.2 File 2: Example that does not require refactoring. We chose our second file, which did not require any refactoring. This was deliberately done to analyse how our tool tackled a code base that has small if-else statements. After executing our tool on the file it is seen that although the code has small if statements it does not refactor the code. However, the indentations and the empty lines are removed by the check file format.py file. Our design makes sure that if there is not any if-else statement that is between six to fifty lines the tool will ignore it.

7.2.3 File 3: Complex Example - I. Our main goal in using this complex example was to find out how our tool performed when it came to code bases that had sophisticated python syntax. Moreover, using these examples helped us in finding edge cases and we tried to resolve as many edge cases we could. However, the most challenging part for us in this example was to find out the parameters that should have been passed into the method that our algorithm extracted. The three important observations that we analysed after applying our tool on file are:

(1) Our tool was successful in finding out the portion of code that needed refactoring.
(2) Two of the methods that our tool extracted are either missing some parameters that should have been passed or extra parameters were passed leading to another sort of code smell of Long Parameter List
(3) After undergoing refactoring it is seen that the line number decreased from 128 to 87.

7.2.4 File 4: Complex Example - II. The main difference in this file compared to the previous one is that it contains a nested for-loop. Moreover, in the nested for-loop there is long if-elif statement with comments in between the if and elif statements. Along with the limitations of the tool that were discussed previously this file intro- duced us with additional problems:

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

8 LIMITATIONS

In our research we were able to detect code smells using ensemble technique and neural network and also proposed a naive approach to refactoring one of the code smells, long method. However, there are limitations to our work that need addressing. Firstly, developers might not address some of the code smells that we proposed because they might find it difficult to address the code smells or the code smells have little to no effect on the code [7]. Secondly, our refactoring algorithm finds any long if-else statement and extracts it for refactoring. However, developers and software engineers have their own perspective on the segment of code that needs refactoring [4]. Addressing perspective is out of scope for our research. Lastly, our algorithm was not able to handle some of the files as discussed in the previous section and also the design of the algorithm is simple which

gives rise to poor performance. The limitations of the algorithm are divided into two subsections, which is discussed below.

8.1 Incomplete Code Refactoring

In the previous section we have analysed in-depth, which kind of code our algorithm cannot handle and why it cannot handle it. In addition to the discussed limitations, there are a few more flaws that have not been addressed yet. It is evident that a long for-loop causes a long method smell. Our tool cannot find the for-loop from given code and refactor it. Moreover, to refactor a python file we import the extracted method from another file. As the extracted method is written in another file it is unaware of any functions that are inside it. Thus, error occurs when we call these extracted methods in place of the actual code.

8.2 Performance Constraints

Our straightforward approach in designing the algorithm gives rises to heavy computations which increases the runtime of the tool if the python files are large. To solve the edge cases we used a lot of if statements, which requires a lot of computation. Moreover, our algorithm finds parts of code that are between a threshold and refactors it. However, this simplistic approach does not take any other factors into consideration thus solves the problem using a greedy approach. The greediness causes the time complexity of the algorithm to be between linear and polynomial.

9 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 [5] from the literature, we present a comprehensive multi-labeled code smell dataset tailored for compatibility with machine learning models. Demonstrating the efficacy of artificial neural networks and ensemble models in accurately detecting code smells, we extend our investigation to the next frontier – automated refactoring of long method using extract method. Employing a basic extract method algorithm, we aim to refactor identified long if-else statement inside a long method, subsequently analysing the results in decrease of code smell in four python files and finding the limitations of our unsophisticated algorithm. Our endeavor underscores the importance of preserving code cleanliness by detecting code smells that requires refactoring and providing a naive approach for refactoring to ensure optimal maintainability in Python code bases.

REFERENCES

[1] Mansi Agnihotri and Anuradha Chug. 2020. A systematic literature survey of software metrics, code smells and refactoring techniques. *Journal of Information Processing Systems* 16, 4 (2020), 915–934.

[2] Francesca Arcelli Fontana, Mika Mäntylä, Marco Zanoni, and Alessandro Marino. 2015. Comparing and experimenting machine learning techniques for code smell detection. *Empirical Software Engineering* 21 (06 2015). https://doi.org/10.1007/s10664-015-9378-4

[3] Lerina Aversano, Umberto Carpenito, and Martina Iammarino. 2020. An Empirical Study on the Evolution of Design Smells. *Information* 11, 7 (2020). https://doi.org/10.3390/info11070348

[4] Sofia Charalampidou, Apostolos Ampatzoglou, Alexander Chatzigeorgiou, Antonios Gkortzis, and Paris Avgeriou. 2017. Identifying Extract Method Refactoring Opportunities Based on Functional Relevance. *IEEE Transactions on Software Engineering* 43, 10 (2017), 954–974. https://doi.org/10.1109/TSE.2016.2645572

[5] Zhifei Chen, Lin Chen, Wanwangying Ma, and Baowen Xu. 2016. Detecting Code Smells in Python Programs. In *2016 International Conference on Software Analysis, Testing and Evolution (SATE)*. 18 – 23. https://doi.org/10.1109/SATE.2016.10

[6] Zhifei Chen, Lin Chen, Wanwangying Ma, Xiaoyu Zhou, Yuming Zhou, and Baowen Xu. 2018. Understanding metric-based detectable smells in Python software: A comparative study. *Information and Software Technology* 94 (2018), 14–29.

[7] Zhifei Chen, Lin Chen, Wanwangying Ma, Xiaoyu Zhou, Yuming Zhou, and Baowen Xu. 2018. Understanding metric-based detectable smells in Python software: A comparative study. *Information and Software Technology* 94 (2018), 14–29. https://doi.org/10.1016/j.infsof.2017.09.011

[8] André C. P. L. F. de Carvalho and Alex A. Freitas. 2009. *A Tutorial on Multi-label Classification Techniques*. Springer Berlin Heidelberg, Berlin, Heidelberg, 177–195. https://doi.org/10.1007/978-3-642-01536-6_8

[9] Seema Dewangaan, Rajwant Singh Rao, Alok Mishra, and Manjari Gupta. 2022. Code Smell Detection Using Ensemble Machine Learning Algorithms. (10 2022). https://doi.org/10.3390/app122010321

[10] Dario Di Nucci, Fabio Palomba, Damian A. Tamburri, Alexander Serebrenik, and Andrea De Lucia. 2018. Detecting code smells using machine learning techniques: Are we there yet?. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 612–621. https://doi.org/10.1109/SANER.2018.8330266

[11] Francesca Arcelli Fontana, Marco Zanoni, Alessandro Marino, and Mika V. Mäntylä. 2013. Code Smell Detection: Towards a Machine Learning-Based Approach. In *2013 IEEE International Conference on Software Maintenance*. 396–399. https://doi.org/10.1109/ICSM.2013.56

[12] Martin Fowler. 2018. *Refactoring: improving the design of existing code*. Addison-Wesley Professional.

[13] Aleksandar Kovačević, Jelena Slivka, Dragan Vidaković, Katarina-Glorija Grujić, Nikola Luburić, Simona Prokić, and Goran Sladić. 2022. Automatic detection of Long Method and God Class code smells through neural source code embeddings. *Expert Systems with Applications* 204 (2022), 117607.

[14] Arun Lakhotia and Jean-Christophe Deprez. 1998. Restructuring programs by tucking statements into functions. *Information and Software Technology* 40, 11 (1998), 677–689. https://doi.org/10.1016/S0950-5849(98)00091-3

[15] Thanis Paiva, Amanda Damasceno, Eduardo Figueiredo, and Cláudio Sant'Anna. 2017. On the evaluation of code smells and detection tools. *Journal of Software Engineering Research and Development* 5 (2017), 1–28.

[16] Jesse Read, Antti Puurula, and Albert Bifet. 2014. Multi-label Classification with Meta-Labels. In *2014 IEEE International Conference on Data Mining*. 941–946. https://doi.org/10.1109/ICDM.2014.38

[17] Jose Reis, Fernando Brito e Abreu, Glauco Carneiro, and Craig Anslow. 2020. Code smells detection and visualization: A systematic literature review.

[18] Rana Sandouka and Hamoud Aljamaan. 2023. Python code smells detection using conventional machine learning models. *Empirical Software Engineering* (05 2023). https://doi.org/10.7717/peerj-cs.1370/supp-1

[19] Mahnoosh Shahidi, Mehrdad Ashtiani, and Morteza Zakeri-Nasrabadi. 2022. An automated extract method refactoring approach to correct the long method code smell. *Journal of Systems and Software* 187 (2022), 111221.

[20] Sagar Sharma, Simone Sharma, and Anidhya Athaiya. 2017. Activation functions in neural networks. *Towards Data Sci* 6, 12 (2017), 310–316.

[21] Tongjie Wang, Yaroslav Golubev, Oleg Smirnov, Jiawei Li, Timofey Bryksin, and Iftekhar Ahmed. 2021. PYNOSE: A Test Smell Detector For Python. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. https://doi.org/10.1109/THS.2011.6107909

[22] Limei Yang, Hui Liu, and Zhendong Niu. 2009. Identifying Fragments to be Extracted from Long Methods. In *2009 16th Asia-Pacific Software Engineering Conference*. 43–49. https://doi.org/10.1109/APSEC.2009.20

[23] Morteza Zakeri-Nasrabadi, Saeed Parsa, Ehsan Esmaili, and Fabio Palomba. 2023. A systematic literature review on the code smells datasets and validation mechanisms. *Comput. Surveys* 55, 13s (2023), 1–48.

[24] Gábor Szo ke, Csaba Nagy, Lajos Jeno Fülöp, Rudolf Ferenc, and Tibor Gyimóthy. 2015. FaultBuster: An Automatic Code Smell Refactoring Toolset. In *2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 253 – 258. https://doi.org/10.1109/SCAM.2015.7335422