Detecting Code Smells in Python Using Ensemble Learning with Advanced Resampling Techniques

Md Jannatul Ferdhous Emon

Bachelor of Science in Computer Science and Engineering

University of Barishal

Class Roll: 19CSE049

Registration Number: 110-049-19

Session: 2018-19

A thesis submitted to the

Department of Computer Science and Engineering, University of Barishal

In partial fulfillment of the requirements for the Degree of

Bachelor of Science in Computer Science and Engineering

University of Barishal

Barishal, Bangladesh

Detecting Code Smells in Python Using Ensemble Learning with Advanced Resampling Techniques

Md Jannatul Ferdhous Emon

Approved:

Signature *Date*

_____ _____

Supervisor: Md. Samsuddoha

_____ _____

Student: Md Jannatul Ferdhous Emon

Abstract

Code smells indicate poor design or potential maintenance issues. Code smells don't cause immediate errors but they can lead to deeper problems over time. They negatively impacted maintenance, performance issues, and increased error rates. Python is becoming more and more popular in several fields, such as artificial intelligence, web development, and data science. Consequently, finding code smells in Python source code is essential. Smelly code smells are rare, hence the datasets for training models are often imbalanced. Also, some code smells depend on specific context. These are the major challenges in the detection of Python code smell. Besides, there is a fewer research based on Python code smells. This gap makes the field highly researchable. To address this issue, researchers are turning to ensemble techniques, machine learning techniques, and methods for resampling datasets.

This research paper's goal is to identify code smells in the Python dataset by combining sophisticated data resampling techniques with ensemble learning approaches. We looked into the most effective ensemble learning techniques for identifying smells in the Python code. Next, we enhance the performance of conventional machine learning models. This analysis is required to improve the prediction performance. In this study, we also apply advanced data resampling methods to address imbalanced datasets. Finally, we analyze the performances between using all features and using feature selection methods. This study shows better prediction accuracy than other relevant research.

This study evaluates 11 different and combined models. We use three ensemble learning models XGBoost, LightGBM, and AdaBoost and two machine learning models Decision Tree, and Random Forest with advanced resampling techniques SMOTE, SMOTEENN. We use two Python datasets to use two types of code smells i.e. Python Large Class code smells and Python Long Method code smells. Each model is based on performance metrics like accuracy and MCC score. Our models include XGBoost, XGBoost with SMOTE, XGBoost with SMOTEENN, LightGBM, LightGBM with SMOTE, LightGBM with SMOTEENN, AdaBoost, AdaBoost with SMOTE, AdaBoost with SMOTEENN, Random Forest with SMOTE and Decision Tree with SMOTE.

This experiment was conducted using two datasets to test each model's classification capacity. According to the findings, the LightGBM with SMOTEENN had the best accuracy for detecting code smells, achieving 98.47% with a 0.97 MCC score for Large Class and 99.79 accuracy with a 1.00 MCC for Long Method. XGBoost with SMOTEENN achieved the highest

accuracy at 99.79 and an MCC score of 1.00 for the Long Method. AdaBoost provides the lowest accuracy at 91.41% and an MCC score of 0.73 for Large Class detection.

The finding suggests that LightGBM with SMOTEENN is the most effective model for code smell detection in Python code both for Large Class and Long Methods. This research attention is focused on detecting code smells on the Python dataset. The study aims to conduct a comparative analysis between ensemble-based models with advanced resampling techniques aiming to help improve the performance of the models for real-world situations. This study shows better performance compared to other relevant work. Future research includes cross-language code smell detection, influencing model interpretability, and more advanced imbalanced data solutions.

Acknowledgments

I would like to thank everyone who helped me finish my thesis. Before anything else, I would like to express my gratitude to my supervisor, **Md. Samsuddoha**, for his exceptional help, tolerance, and guidance throughout the thesis. His perceptive comments, helpful critiques, and recommendations have been crucial in helping to improve and shape the thesis.

In addition, for their comprehension, encouragement, and support during the thesis, I would want to thank my family. They have been a constant source of inspiration for me because of their unfailing support and belief in my abilities, for which I am honestly grateful.

I also express my heartfelt appreciation to all the software engineers, researchers, and individuals whose dedication to advancing software development and technology inspires and drives our collective efforts toward improving software quality and outcomes.

Finally, I want to convey my appreciation to the Computer Science and Engineering faculty members of the University of Barishal for their outstanding instruction and knowledge, which have been crucial to the successful completion of this thesis.

Thank You

Md Jannatul Ferdhous Emon

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Context

Code smell is a portion of the source code that does not affect software features but can negatively affect code quality, and maintenance and lead to deeper problems over time. It is not like regular bugs or does not affect the functionality of the software. It indicates the weaknesses of the design. It also can reduce the development speed and raise the possibility of future failures. Because code smells are early warning signs of potential problems that could occur in the codebase, they are essential indicators of the state of the software. They are crucial because, even if the code is functioning well right now, they can spot potential weak points that could lead to issues down the road. By identifying code smells, developers can stop basic design defects, maintainability problems, and scalability problems before they worsen. This makes it crucial to detect smells in source code.

However, detecting smells manually is very difficult. For this reason, researchers are looking into methods for automatically identifying code smells in source code. By employing Machine Learning (ML) and Deep Learning (DL) approaches, researchers demonstrate a notable improvement in code smell detection. Code Understanding BERT (CuBERT), cove2vec, and code2seq provide rich features to improve detection accuracy [4]. Most of the datasets in code smell detection are imbalanced with fewer ‘smelly’ samples. Advanced resampling techniques like Synthetic Minority Oversampling Technique (SMOTE) [27] and Synthetic Minority Oversampling Technique with Edited Nearest Neighbors (SMOTEENN) [28] are applied to address the class imbalance and enhance performance. Ensemble methods also effectively improved the detection robustness. The transferability of code across datasets and programming languages is one of the main issues with code smell detection. Smells on different programming language datasets are not detectable by the model that was trained on the dataset for one programming language.

In the late 1990s, Kent Beck popularized the term "code smell" on WardsWiki. The phrase first appeared in Martin Fowler's 1999 book "Refactoring: Improving the Design of

Existing Code" [26]. Following it, the term's usage increased. Agile programmers also used the term. The main goal of refactoring the code smells is to increase the code maintainability along with its understandability. Martin Fowler said that,

> "Any fool can write code that a computer can understand. Good programmers write code that humans can understand."

1.2 Problem Statement

One of the main limitations of Python code smell detection study is its application. The problem statement is to identify the ensemble models that provide outstanding performances in Python code smell detection. Considering the effectiveness of different models, this research provides valuable information for detecting Python code smells in real-world applications. This research addresses the critical requirement for the evaluation of ensemble-based models in Python code smell detection tasks. Existing research has a huge gap in dataset size and smell detection performance. There are very few samples in the majority of Python code smell datasets with fewer or no 'smelly' samples. The performances of these researchers are comparatively lower. We select a dataset with a predetermined number of samples for both smelly and non-smelly to fill in these gaps. Next, we plan to evaluate and contrast typical ML models with those that incorporate the SMOTE with traditional ML models. After that, we included some Ensemble models like Extreme Gradient Boosting (XGBoost) [29], Light Gradient-Boosting Machine (LightGBM) [30], and Adaptive Boosting (AdaBoost) [31] to enhance our study's scope. Through this research, we aim to help improve Python code smell detection performance for real-world situations.

1.3 Research Objectives

The majority of previous studies on code smell detection were on code smell detection based on the Java programming language. This sector covers traditional ML to advanced DL algorithms to identify smells accurately. Even some research has introduced advanced resampling techniques to do this task. However, a small number of researchers presented Python code smell detection techniques in their research. Therefore, more research is required to examine the performance of code smell detection in Python programming language. This study's primary goal is to detect Python code smells by using ensemble learning models. Additionally, we worked to enhance Python code smell detection's performance and address

imbalance datasets using advanced resampling techniques in this study. Finally, we assess how feature selection techniques affect prediction accuracy.

1.4 Research Question

In our research, we are looking into the following research inquiries:

RQ1. Which ensemble learning algorithm is better for detecting Python code smells?
This question analyses the comparative effectiveness of ensemble learning algorithms like XGBoost, LightGM, and AdaBoost for detecting code smells in Python datasets. It looks to figure out which model delivers a more effective performance in Python code smell detection.

RQ2. How can we improve the performance of ML models in Python code smell detection?
This inquiry explores the effectiveness of several ML models and examines ways to enhance their performance. This provides important insights into improving the prediction performance of traditional ML models.

RQ3. Does the SMOTE and SMOTEENN class balancing techniques improve the performance of Python code smell detection?
SMOTE and SMOTEENN are the latest and most advanced data resampling techniques. This question aims to reveal the impact of SMOTE and SMOTEENN on model performances. Additionally, it offers useful details on the advantages and disadvantages of various datasets and models.

RQ4. Does the feature selection method affect the performance of the prediction?
The purpose of this inquiry is to determine how the feature selection process affects model performance to detect Python code smell detection. By this question, we can understand whether we should use selected features or all the features from the set of features.

1.5 Significance of the Study

The research Detecting Code Smells in Python Using Ensemble Learning with Advanced Resampling Techniques is expected to make several valuable contributions to the

software quality assurance field. This study aims to improve detection accuracy by handling class imbalance of datasets. This can lead to more robust and reliable Python code smell detectors. This study also contributed to a better understanding of which features are most effective for identifying code smell in Python. Developers and Software Engineers can identify problematic code sections, reduce the accumulation of technical debt, and enhance software quality with automated detection methods. Other researchers can develop more robust models, and investigate cross-language transferability by inspired from this research. In real-world scenarios, our findings can be integrated into Integrated Development Environment (IDE) Plugins, Automated Refactoring Tools, Continuous Delivery or Deployment (CD) Pipelines, and Code Review Tools. We encourage future researchers for cross-language code smell detection, influencing model interpretability and more advanced imbalanced data solutions.

1.6 Contributions

In this study, we utilize the ensemble learning models with data resampling techniques to detect Python code smell detection for the first time and achieved an outstanding performance. Additionally, for the first time, we combine sophisticated data resampling methods with conventional ML models, such as Random Forest (RF) and Decision Tree (DT), to detect code smells. Among the ensemble learning models, XGBoost performs better than the models proposed in previous studies without any resampling. After resampling, all the ensemble models show better improvement in performance. The XGBoost and LightGBM achieved the highest performance with data resampling. It is also investigated that the ML models perform better with advanced resampling techniques like SMOTE. This was our initial aim. Then we investigated whether the advanced data resampling techniques improved the models' performance. In this study, we discovered that practically all models' performance is enhanced by the SMOTE and SMOTEEN class balancing strategies. With SMOTEENN, we achieved the highest forecast accuracy. Additionally, we looked at feature selection techniques and how they affected the effectiveness of identifying Python code smells. We found that some models e.g. XGBoost, DT, RF perform well considering all the features. Other models perform well only with SMOTEENN considering all features. Through our contributions, we seek to advance the understanding and application of the ensemble-based models in Python code smell detections, paving the way for more reliable and adaptable solutions in real-world applications.

1.7 Structure of Thesis

Each of the distinct chapters that make up this thesis format examines a different facet of the study. An Introduction, Literature Review, Methodology, Results and Discussions, and Conclusion are among the chapters.

In summary, Research attention is focused on detecting code smells on the Python dataset. The study aims to conduct a comparative analysis between ensemble-based models with advanced resampling techniques aiming to help improve the performance of the models for real-world situations. Each chapter discusses specific aspects of the research, providing a methodology, finding the result, and discussing key findings.

Chapter 2

Literature Review

Sandouka and Aljamaan presented a new Python code smell dataset targeting Large Class and Long Method code smells [1]. Eighteen features were retrieved from the source codes, and 1000 examples of each type of code smell were prepared for this dataset. Six conventional ML methods are assessed by the authors as baselines for identifying Python code smells. RF, DT, Support Vector Machine (SVM), Logistic Regression (LR), Multi-Layer Perceptron (MLP), and Stochastic Gradient Descent (SGD) are some of the models used for classification challenges. The RF ensemble model performed best in identifying Large Class code smells, with an accuracy of 92.7 and a Matthews Correlation Coefficient (MCC) rate of 0.77. However, the MCC and accuracy values of 95.9 and 0.90, respectively, are the greatest, and the DT model was the most effective at identifying Long Method code smells. That means RF is strong in one aspect, while probably DT might fit some other types of code smells. Only six traditional ML models were evaluated in the study. Other advanced algorithms and techniques, including DL models, may work even better. Excluding such models can lead to a lack of better proposals for code smell detection.

The performance of several ML models in predicting code smell for diverse datasets was examined by the authors of this work [2]. The authors discussed how feature selection methods affect ML models' performance. Six ML algorithms—Naive Bayes (NB), K-Nearest Neighbor (KNN), MLP, DT, RF, and LR—were used by S. Dewangan et al. to detect code smells. In this work, each has been evaluated for performance against the following code smells: God-class, Data-class, Feature-envy, and Long-method. When paired with the chi-square feature selection method, the God class detects it with an accuracy of 98.21%. When RF runs on Data-class, after taking all the features in, it returns 99.74% accuracy. When applied to the entire Long-method dataset, logistic regression will yield a 100% accuracy rate and a 100% F1 score. When all the features were taken into account, the DT method performed best on the Feature-envy dataset, with a maximum accuracy of 98.60%. The dataset used by the authors was obtained directly from a study published by Fontana et al. [3]; hence, there could be some kind of bias or limitation specific only to that dataset. Also, the dataset has been purely

extracted from Java source code and hence there are issues with the external validity of the results. That may also mean that the nature of other languages, like Python/C++, differs considering characteristics and patterns of code smell.

To detect code smells, the suggested study [4] evaluated the effectiveness of contemporary embeddings such as code2vec, code2seq, and CuBERT against the goodness of standard code metrics. That is an important investigation because it forms the basis for the representation that should be used in future models for detection. Besides, this study investigated the possibility of transfer learning, i.e., whether code smell detection tasks can benefit from the information gained from code understanding models. As this was an area not much covered, by evaluating several neural pre-trained source code embeddings in terms of their applicability to the code smell detection task, the authors sought to close that gap. To evaluate ML Denotements in detecting Long Method and God Class code smells in this study, A. Kovacevic et al. used RF, SMOTEENN, Bootstrap Aggregating (Bagging), SMOTE, XGBoost, and Heuristic-based Approaches. When paired with traditional code metrics, neural embeddings—code2vec, code2seq, and CuBERT—perform better than all other source code representations. However, the authors failed to provide the way specifically in which the effectiveness is measured. Where the metrics were not clearly stated, it would be hard to establish the performance difference among various models and methods of study.

The research study [5], which investigated the efficacy of ML approaches on the severity classification of code smells, measured the performance of multiple ML models for the classification of code smell severity. Among the ML methods the authors employed were Binary Classifiers for Ordinal Classification, Regression Techniques, and Multinomial Classification, to classify the intensity of code smells. Fontana et al. concluded that while the binary classification's accuracy in determining whether a code smell is present or not was higher than the classification's accuracy in determining the intensity of code smells, it was still very remarkable for the best models in their research. The minimum and maximum Spearman Ranking Correlations range from 0.88 to 0.96, demonstrating that the ranking of actual and expected severity levels is highly correlated. The authors reported a certain number of performance measures; however, definitely, this cannot be considered an exhaustive analysis of all the possible metrics that can be relevant for such models. It lessens comprehension of the model's advantages and disadvantages given several classification aspects.

The research [6] estimated the performance of some ensemble methods to get the best method that gave the best accuracy of bug report severity prediction. They also analyzed a pre-processing technique to see the effect of the pre-processing techniques on prediction accuracy. Researchers used Bagging, Voting, Adaboost, and RF in their experiment. Among all the other ensemble algorithms, Bagging can outperform them in terms of accuracy in this experiment. While the accuracy of the different ensemble methods for PitsA ranges between 58.09 and 74.07, it does between 54.38 and 80.72 for PitsB, between 79.85 and 89.80 for PitsC, between 93.42 and 96.20 for PitsD, between 69.21 and 72.36 for PitsE, and finally between 64.10 and 75.70 for PitsF. Pushpalatha and Mrunalini channeled their work toward performance studies concerning the accuracy evaluation metric. While accuracy is important, additional measures could provide a more comprehensive picture of the model's performance. F1-score, recall, and precision are some of these additional metrics.

Five Python code smells were examined by Vatanapakorn et al. [7] i.e. Long Method, Large Class, Long Parameter List, Long Base Class List, and Long Scope Chaining taking classes and functions into consideration. The authors determined the most pertinent elements that helped detect these kinds of smells and tried applying a variety of ML techniques to find these code smells on Python codebases. Vatanapakorn et al. studied and implemented eight ML techniques: DT, Gradient Boosted Trees (GBT), RF, SVM, KNN, LR, MLP, and NB. The models have achieved 99.72% accuracy for a few code smells, such as Long Method and Long Base Class List. The precision, recall, and F1-score for Long Method detection were 96.43%, 100.00%, and 98.18%, respectively, while the classifiers' results for Long Scope Chaining identification were 100.00%, 95.83%, and 97.87%. Except for MLP, all the ML classifiers of the code smell of the Long Base Class List produced the same values: 96.88% for precision, 100.00% for recall, and 98.41% for F1-score. The dataset used by the Authors contains a very small amount of data and is highly imbalanced e.g. Long Method contains only 2 positive tuples and Long Base Class List doesn’t contain any positive tuple in the training dataset. This can negatively impact the generalization and detection of the model corresponding to those smells.

The best ML models for identifying method-level code smells in switch statements and long parameter lists were examined by Dewangan and Rao in this study [8]. Most of the time, instances of code smell are comparably few to those without smells. This may be due to a class imbalance in the datasets used for training models. It has been solved by Dewangan and Rao

using the SMOTE technique. The authors reported a maximum of 97.12% from a long parameter list dataset measured by a max voting ensemble method.

This work [9] employed two DL algorithms and five ensemble ML techniques to identify code smells. XGBoost, Gradient Boosting (GB), Max Voting, AdaBoost, and Bagging are the ensemble learning methods that are employed. The problem of class imbalance is addressed by the SMOTE class balancing technique. The authors used four different code-smell datasets: Data class, God class, Feature-envy, and Long-method. Using the Long-method and Feature Envy datasets, 100% accuracy was achieved for all five ensemble models. The Bagging method alone performed best on the God Class dataset with an accuracy of 99.24%, while the Max voting, GB, and XGBoost methods performed best on the Data Class dataset with 100% accuracy each. The techniques developed by S. Dewangan et al. may not be suitable for all programming languages. Specifically, the authors note that their approach may not apply to Python and C++ programming languages, which could restrict the broader applicability of their findings to other software development environments.

The study [10] detailed how well ML techniques performed in code smells detection. The authors believed that there was a dire need for a comprehensive evaluation of these algorithms to understand their strengths and weaknesses. SVM, NB, Decision Tree C4.5 (J48), RF, and Ensemble Learning are among the ML techniques utilized in the code smell detection challenge. This review attempts to locate and gather relevant research conducted between 2005 and 2024 to provide a broad overview of the application of ML techniques for code smell detection. 42 pertinent linked research publications were found and critically examined by Yadav et al., who also demonstrated that several ML algorithms were used in code smell identification. These indicate several different efficacies, with especially ensemble learning methods featuring in 21 of the analyzed studies. When evaluating the effectiveness of ML algorithms, performance measurements such as accuracy and area under the ROC curve (AUC ROC) are employed. These are quantitative metrics; thus, they give the extent these algorithms do identify code smells, hence showing that in certain contexts some algorithms perform better. The dataset of all relevant studies is coded in C#, JAVA, and XML. That would imply the severity of the deficiency in studying either the Python or C++ language.

Talaya and Joachim compared five distinct ML algorithms to identify vulnerabilities in source code in their study [11]. Python source code was the authors' primary emphasis. Among

the methods used in their investigation are Gaussian Naive Bayes (GNB), DT, LR, MLP, and Bidirectional Long Short-Term Memory (BiLSTM). By using and comparing these methods, Talaya and Joachim were able to identify seven categories of vulnerabilities, including SQL injection, remote code execution, command injection, cross-site scripting (XSS), path disclosure, cross-site request forgery (XSRF), and open redirect. With an average of 98.6% Accuracy, 94.7% F-Score, 96.2% Precision, 93.3% Recall, and 99.3% ROC—the highest score of any model—the BiLSTM model with word2vec Embeddings demonstrated exceptional performance. It shows that advanced ML and ensemble models provide more accurate predictions.

The research study [12] implemented a DT to analyze the pattern of Python code indicating the smell. The Authors have pinpointed that detection requires a structural approach toward analysis and detection. They also measured the performance of DT algorithms in spotting code smells. The authors employed J48 to identify code smells in the Python dataset. It is also used for pre-processing the data and choosing features in their study. The performance matrix of the J48 algorithm for the resultant dataset was as high as 100% regarding accuracy, recall, and F1-score, meaning the model will analyze code smells in Python code perfectly. One serious limitation is that there are only 147 data points that the authors have used to train their model. This might make the model less robust. In particular, the model's robustness and performance will change with the dataset size.

All of these studies have shown modern approaches to identifying smells in programs through ML models. Even many of them used modern and advanced ML models for better efficiency. However, the majority of the available datasets for code scent identification are based on the Java programming language, and the majority of the studies concentrated on Java code smell detection. Very few researchers have discussed and tried to detect code smells of Python and C# programming language. As far as we are aware, there are only three studies have been published from 2022 to 2024 based on Python programming language. Among them, one is about detecting source code vulnerability, and the rest are about detecting code smells. In their investigation, Talaya and Joachim examined five distinct ML techniques to identify source code vulnerabilities [11]. The dataset they used contained only 147 data points which may make the experiment less robust. Vatanapakorn et al. [7] discussed five code smells of the Python codebase and implemented eight ML algorithms to identify those smells. Even though the authors increased the size of the dataset some smells got a very small number of positive

tuples e.g. Long Method contains only 2 positive tuples and Long Base Class List doesn't contain any positive tuples in the training dataset. This may produce a high false negative and reduce the accuracy of the algorithms. To target Large Class and Long Method code smells, Sandouka and Aljamaan [1] created a new Python code smell dataset with 1000 samples for each type of code smell, of which 20% are positive and 80% are negative tuples. They got the best result for Large Class code smell detection performance for the RF ensemble model, with 92.7 accuracy and the DT model for detecting Long Method code smells with an Accuracy of 95.9%. Even though they have generated a good result this is still not the best for real-world smell detection scenarios. In our study, we analyzed the modern and advanced ML models and used them to identify Python code smells more accurately.

Chapter 3

Methodology

The methodology of Detecting Code Smells in Python using Ensemble Learning with Advanced Resampling Techniques has been outlined. In this section, we discuss the data collection process, selected code smells, how we select models, and run every model with different datasets. We describe our materials and models in this section. And also describes the models' execution process and evaluation metrics.

3.1 Data Collection

In this study, we train and assess our models for Python code smell detection using two datasets. The datasets used by us were constructed and used by Sandouka and Aljamaan [1].

The first dataset is about Python Large Class code smell. The second dataset is about Python Long Method code smell. Both datasets contain two classes: smelly and non-smelly. Each dataset contains 1000 instances. The samples are 20% smelly and 80% non-smelly. There are eighteen different aspects for both kinds of smells. Program length, program vocabulary, volume, calculated program length, effort, difficulty, number of delivered bugs, number of lines of code, number of logical lines of code (LLOC), number of lines of code (LOC), number of comment lines, number of source lines of code (SLOC), number of blank lines, number of lines representing multi-line strings, number of distinct operands, number of distinct operators, number of operands, number of operators, and time required to program are some of these features. These features are classified into two types: Raw metrics feature and Halstead complexity metrics feature. Table 3.1 represents the features based on their types. To the best of our knowledge, these datasets represent the biggest collections of Python code smells. These are valuable resources for developing robust and accurate algorithms that can identify code smells in Python source code.

Metric Type	Raw Metrics	Halstead Complexity Metrics
Metrics	number of logical lines of code (LLOC), number of lines of code (LOC), number of comment lines, number of source lines of code (SLOC), number of blank lines, number of lines that represent multi-line strings	number of distinct operands, number of distinct operators, total number of operands, total number of operators, program vocabulary, program length, calculated program length, volume, effort, difficulty, number of delivered bugs, time required to program

Table 3.1: The code metrics feature for each type of smell

3.2 Code Smells Selection

Code smells are parts of source code that may not directly damage the system but can make it more difficult to read and maintain. Code smells are not bugs or errors. It indicates the deeper problem of any source code. It may also indicate the weakness of the system design that can lead to poor design. A portion of smelly code can reduce the robustness of the source code and the whole system. Programming languages used to write the source code can have different types of code smells.

Various kinds of code smells have been identified in the Python programming language. Eleven categories of Python code smells were introduced by Chen et al. [13]. The code smells that we selected were Large Class and Long Method.

- Large Class: A class-level code smell that has a lot of parameters, properties, methods, or objects with a lot of lines of code is called a large class code smell.
- Long Method: A long method code smell is a method-level code smell that is difficult to grasp, has a complex structure, and includes nearly all of a class's logic.

We chose these two smells because developers most frequently create these code smells. Due of their significant detrimental effects on developer performance and software quality, they are also the most researched smells in other programming languages.

3.3 Models Selection Approach

The major criteria for selecting ensemble-based ML models are the following:

1. Searching from popular websites like IEEE Xplore, Google Scholar, X-mol, Science Direct
2. Select research papers from 2016-2024 based on code smell detection with ML and DL
3. Select papers with more number of citation
4. Select papers with an Impact Factor (IF) rating greater than 3
5. Accuracy of more than 90%
6. Overfitting potentiality and class imbalance handling
7. Choose two popular ML models that perform well to detect code smells
8. Model diversity and performance gain

3.4 Selected Models

In this research, to find Python code smells in Large Class and Long Method, we employ eleven ML models. We use three ensemble learning approaches and oversampling imbalanced classification techniques. We combine them in different ways to produce better performance. This section discusses all the models we have used in this experiment.

3.4.1 XGBoost

Extreme Gradient Boosting is referred to as XGBoost. It is a distributed gradient-boosting library that has been optimized. It is incredibly portable, effective, and adaptable. XGBoost offers parallel tree boosting and applies ML techniques under gradient boosting. Managing the dataset's missing values is quite effective. It also can train a large data in a reasonable time. DT are generated sequentially in this approach. XGBoost relies heavily on weights. Before being added to the DT, which generates predictions, each independent variable is assigned a weight. The second DT receives variables that the first one mispredicted and gives them a larger weight. In Figure 3.1, we show the Schematic illustration of XGBoost. A robust and more accurate model is then produced by combining these separate classifiers.

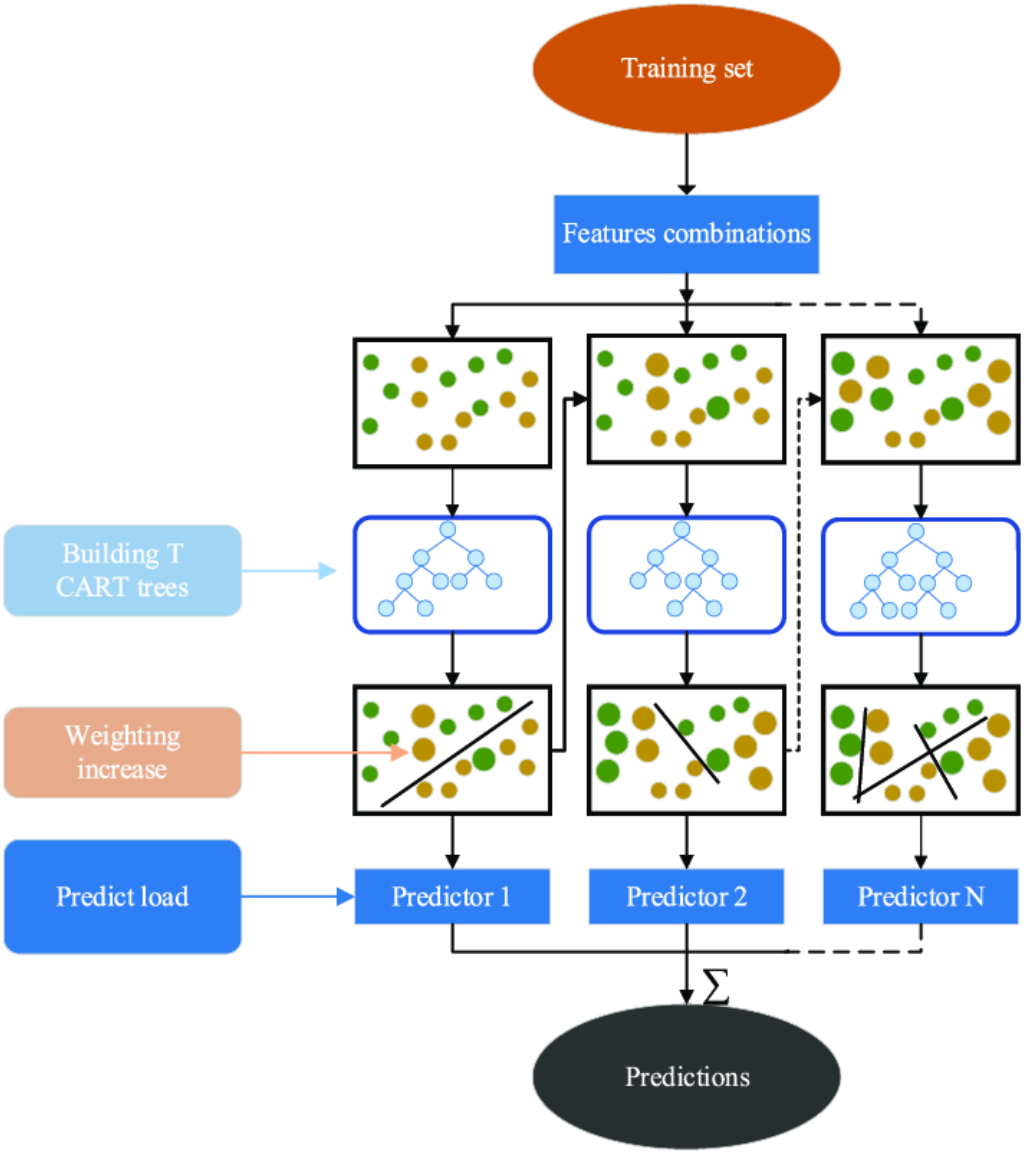

Figure 3.1: Schematic illustration of XGBoost [14]

In most of the classification datasets, there are given some input features and a target feature. The XGBoost algorithm extracts all of the features from the dataset. Then it calculates the pseudo residuals by subtracting the targeted feature from the default initial preds. Then comes to the main part of the algorithm. The algorithm starts with all residuals in the same leaf. The similarity score has been calculated using the following formula:

$$\text{Similarity Score} = \frac{(\sum residuals)^2}{\sum p_{i-1}*(1-p_{i-1})+\lambda} \quad (3.1)$$

Then it selects the features sequentially from the beginning. Each time it sorts the values of the selected features and further goes through the values. The algorithm takes two values at a time and gets the mean of these two values. The leaf residuals are divided according to these

mean values. Residuals of elements with more than the feature's mean value to one node and others to a different. The algorithm then finds the split with the Gain value.

Gain = ss(left_child) + ss(right_child) – ss(Root) (3.2)

Through this process, XGBoost split the node and built the tree. Usually, the stopping of splitting is max depth or maximum number of residuals in the leaf. It is calculated using cover which is defined as the difference between the denominator of similarity score and lambda.

Cover = $\sum p_{i-1} * (1 - p_{i-1})$ (3.3)

By default, the value of the cover is set to 1 and the algorithm discards the division when the value reaches less than 1. Also, if the hyper-parameter gamma (r) value is greater than the Gain of a parent node then the algorithm prunes its children node. After constructing each tree, the algorithm calculates the output value for each leaf node using the following formula:

Output Value = $\frac{(\sum residuals)}{\sum p_{i-1}*(1-p_{i-1})+\lambda}$ (3.4)

This was all for a single tree. In XGBoosting, it creates multiple trees and combines them to generate better performance. To do that, the algorithm converts the initial probability into log(odds) and converts it back to probability. The odds are the ratio of something happening. The steps are:

$\frac{p}{1-p}$ = odds (3.5)

log($\frac{p}{1-p}$) = log(odds) (3.6)

Probability = $\frac{e^{\log(odds)}}{1+e^{\log(odds)}}$ (3.7)

This is the scenario for a single iteration for classification. The performance of XGBoost models can be determined using the loss function. The structure of the loss function is:

Loss = $[\sum_{i=1}^{n} L(y_i, p_i^0 + O_{value})] + \frac{1}{2}\lambda O^2_{\text{value}}$ (3.8)

XGBoost uses a greedy algorithm to build trees and split nodes using only the gain value.

3.4.2 XGBoost with SMOTE

The code smells dataset contains both positive and negative tuples. However, the datasets may contain very few numbers of positive or smelly tuples compared to negative or

non-smelly tuples. This can lead to an imbalanced dataset. It can be difficult to work with imbalanced datasets since most ML methods ignore the minority class, which leads to less-than-ideal results. The minority class's performance is more important in code smell detection, though. One method for dealing with unbalanced datasets that oversamples the minority class is the SMOTE. This is a very simple and easy approach. It increases the number of minority class samples by duplicating them. Figure 3.2 displays the SMOTE simulation plot.

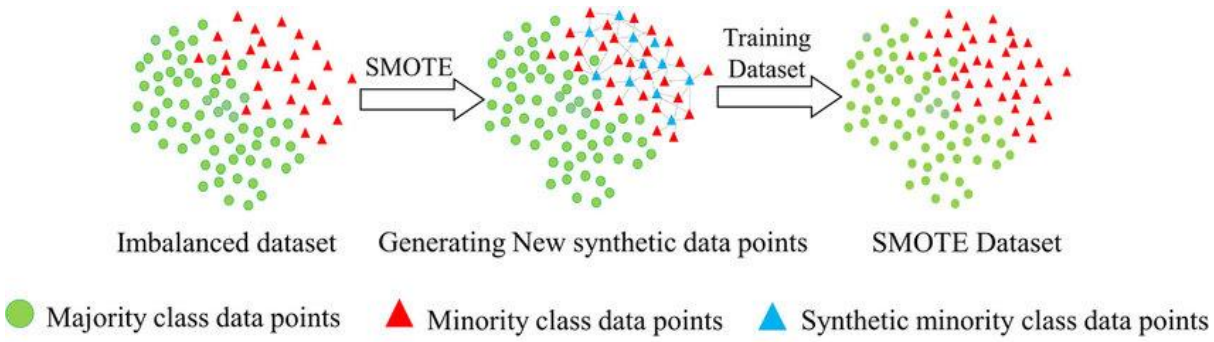

Figure 3.2: Simulation Plot of SMOTE [15]

Several ML algorithms use the SMOTE to handle unbalanced datasets and enhance algorithm performance. It first determines whether a class in the dataset is a minority. Next, the SMOTE chooses the minority class and uses the KNN algorithm to determine the class's closest neighbors. It draws a straight line to connect the selected samples to each of its nearest neighbors. After that, a new point is placed on each line based on a random scaling factor. SMOTE repeats the process till the desired number of synthetic samples is generated. Once the dataset is synthesized, it is ready to train the XGBoost algorithm. The algorithm is trained using the synthesized database for better performance. The architecture of XGBoost with SMOTE is shown in Figure 3.3. XGBoost is a scalable distributed gradient-boosted decision tree (GBDT) ML technique. It provides parallel tree boosting and is the best ML method for problems like regression, classification, and ranking. It is incredibly portable, adaptable, and efficient. It offers parallel tree boosting and applies ML algorithms under GB. Additionally, it is quite effective at handling the dataset's missing values.

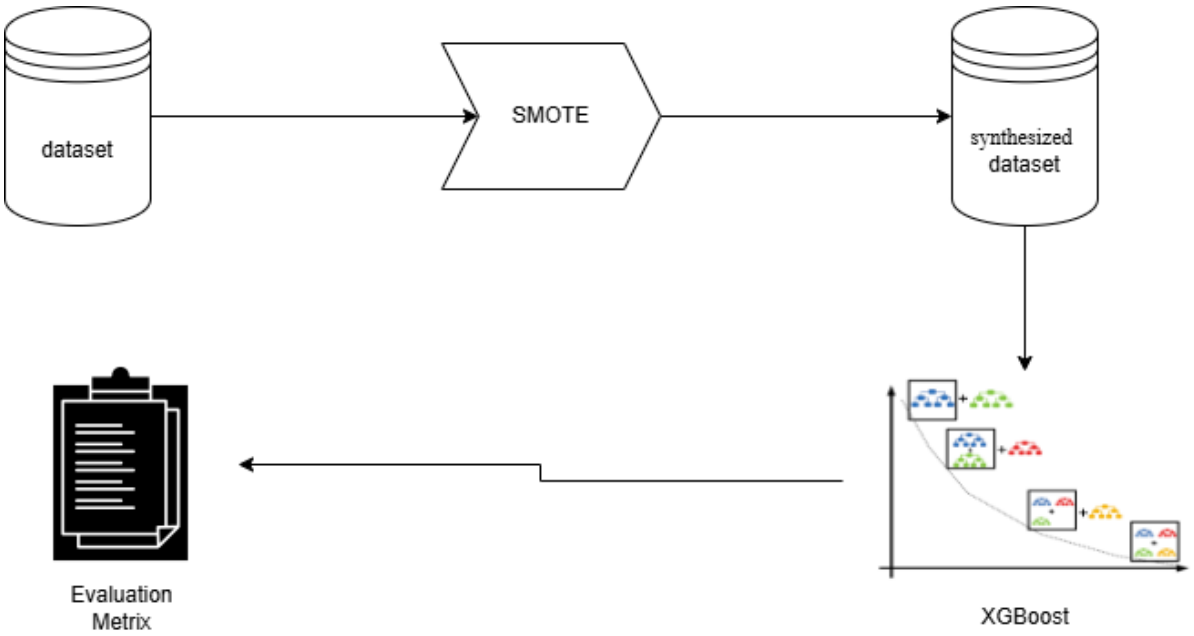

Figure 3.3: Architecture of XGBoost with SMOTE

3.4.3 XGBoost with SMOTEENN

SMOTEENN is a combination of SMOTE and Edited Nearest Neighbor (ENN) algorithms. The most sophisticated technique for handling and resolving unbalanced datasets in the classification model is this one. Either the dominant class is under-sampled or the minority class is oversampled. The performance of several ML models is enhanced by this technique. The ENN approach starts by determining the K-nearest neighbor of each observation. It then determines if the class of the observation and the majority class of its k-nearest neighbor are the same. If the classes of the observation and its K-nearest neighbor are different from the majority class of the observation's K-nearest neighbor, the observation and its K-nearest neighbor are eliminated from the dataset.

SMOTE's capacity to create synthetic instances for minority classes and ENN's ability to exclude certain observations from both classes that are determined to belong to a different class than the observation's class and its K-nearest neighbor majority class are combined in the SMOTEENN technique. Using this strategy, the SMOTE randomly chooses data from the minority class. Next, the distance between the K-nearest neighbors and the random data is calculated. A random number between 0 and 1 is applied to the variants. The multiplication result is then added to the minority class as a synthetic example. This method is continued until the necessary number of minority classes are created. ENN chooses a number K as the number of nearest neighbors after the target percentage of the minority class is reached. ENN delivers

the majority class from the K-nearest neighbor of the selected observation after determining the K-nearest neighbor of the selected observation from the remaining observations in the dataset. Both the observation and its K-nearest neighbor are removed from the dataset if their classes are different from the majority class of their neighbors. Until the required percentage of each class is reached, this process is repeated. Figure 3.4 represents the simulation plot of the whole process.

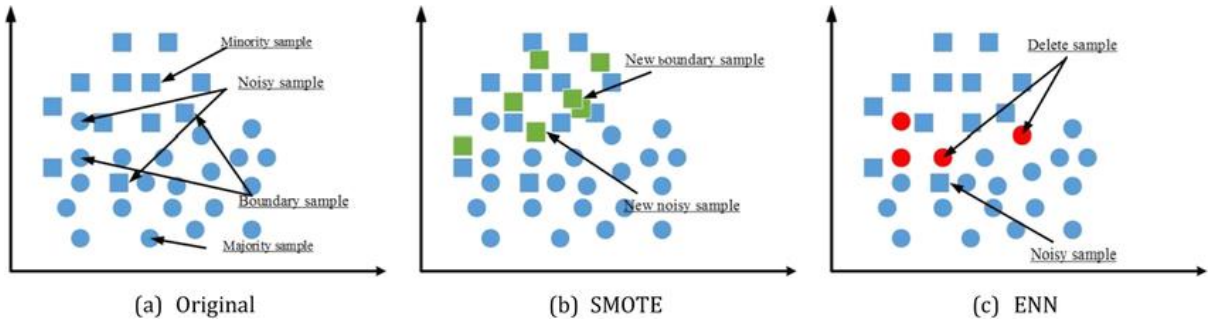

Figure 3.4: Simulation Plot of SMOTEENN [16]

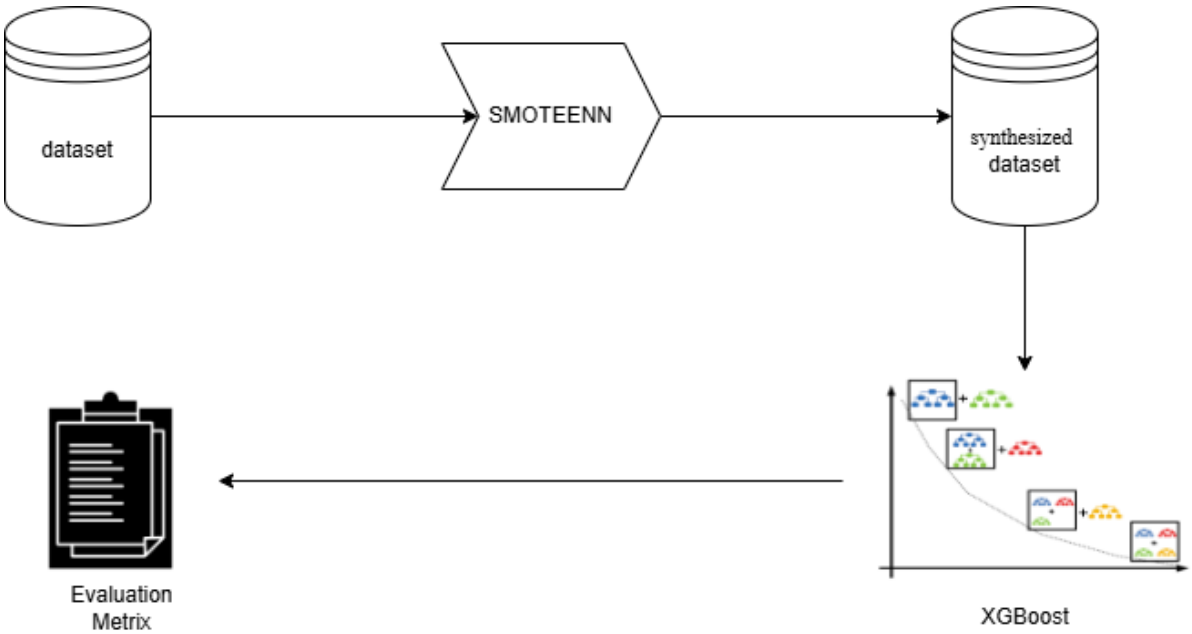

Figure 3.5: Architecture of XGBoost with SMOTEENN

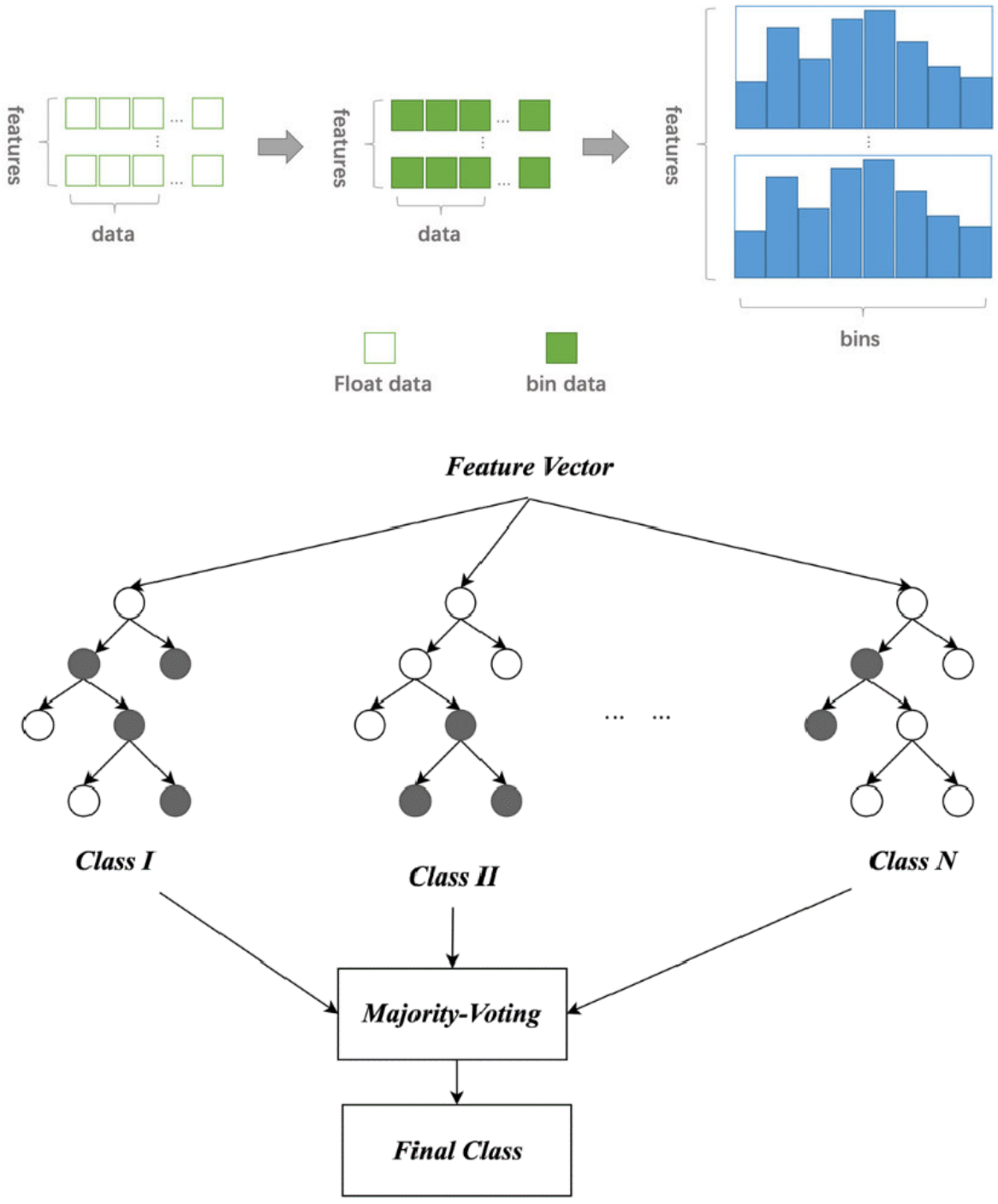

Figure 3.6: Architecture of LightGBM [17] [18]

XGBoost algorithm is trained using the database that is synthesized by the SMOTEEN method. The architecture of XGBoost with SMOTEENN is shown in Figure 3.5. XGBoost, is a scalable distributed GBDT ML technique. It provides parallel tree boosting and is the best ML method for problems like regression, classification, and ranking.

3.4.4 LightGBM

Tree-based learning techniques are used in the GB method LightGBM. It can handle large-scale datasets, is extremely efficient, and has a faster training speed. It provides more accurate predictions than other traditional ML algorithms. It also requires lower memory and supports parallel, distributed, and GPU learning. LightGBM is also known as an ensemble learning framework.

DT are created by LightGBM that grow leaf-wise. Depending on gain, only a single leaf can split based on the given condition. There are two types of leaf-wise tree growth: greedy growth and asymmetrical trees. Greedy growth focuses on the most fruitful path while asymmetrical trees target maximizing accuracy with fewer resources. Limiting the tree's depth can help prevent overfitting due to smaller datasets. LightGBM uses the histogram method. In the histogram method data is bucketed into bins using a histogram of the distribution instead of finding the split points. The bin number can be handled by max_bin_my_feature, max_bin, bin_construct_sample_cnt, and min_data_in_bin. These bins are used to calculate, split the data, and calculate the gain. The dataset in LightGBM is sampled by Gradient-based One Side Sampling (GOSS). GOSS weights data points with a larger gradient higher when calculating the gain. By eliminating points with low prediction errors, GOSS lowers the sample size and enables attention to be paid to locations with high prediction errors. In Figure 3.6 we represent the architecture of LightGBM.

LightGBM determines variance gain for every leaf node after constructing bins for histograms and exclusive feature bundling. Additionally, it determines which node is optimal to split, which greatly enhances performance. The following formula is used to determine the variance gain.

$$\tilde{v}_j(d) = \frac{1}{n}\left(\frac{\left(\sum_{x_i \epsilon A_l} g_i + \frac{1-a}{b}\sum_{x_i \epsilon B_l} g_i\right)^2}{n_l^j(d)} + \frac{\left(\sum_{x_i \epsilon A_r} g_i + \frac{1-a}{b}\sum_{x_i \epsilon B_r} g_i\right)^2}{n_r^j(d)}\right) \quad (3.9)$$

Where,

- d = splitting criteria
- n = total points
- j = features for applying splitting
- b = small gradient data sampling ratio
- a = large gradient data sampling ratio
- A_l = points with a high gradient that are less than or equal to d

- B_l = points with a low gradient that are less than or equal to d
- g_i = gradient for i-th point
- n_l = points in B_l + points in A_l
- A_r = points with a high gradient that are greater than d
- B_r = points more than d and has a low gradient
- N_r = points in B_r + points in A_r

To select the finest examples in this technique, GOSS first sorts the data points according to the gradients' absolute values. The instances are then chosen at random from the remaining data. GOSS then amplifies the sampled data with minor gradients by a constant to compute information gain. This information is subsequently passed on to the following weak learner. Each feature in node i will have its variance gain determined by LightGBM, which will then choose the feature that splits the best and yields the maximum variance gain. The maximum variance gain across all leaf nodes will be compared when a feature that produces the highest variance gain for each leaf has been chosen, and the leaf node with the highest variance gain will be divided. After repeating this leaf-splitting process until the desired result is achieved, residuals are computed, a new input set is constructed using the residuals and the expected value, and the data is then transferred to the following tree. Lastly, LightGBM aggregates each tree's output, just like other GBDT methods do.

3.4.5 LightGBM with SMOTE

Several ML algorithms use the SMOTE to handle unbalanced datasets and enhance algorithm performance. It first determines whether a class in the dataset is a minority. Next, the SMOTE chooses the minority class and uses the KNN algorithm to determine the class's closest neighbors. It draws a straight line to connect the selected samples to each of its nearest neighbors. After that, a new point is placed on each line based on a random scaling factor. SMOTE repeats the process till the desired number of synthetic samples is generated. Once the dataset is synthesized, it is ready to train the LightGBM algorithm.

LightGBM generates DTs that grow leaf-wise. Depending on gain, only a single leaf can split based on the given condition. There are two types of leaf-wise tree growth: greedy growth and asymmetrical trees. Greedy growth focuses on the most fruitful path while asymmetrical trees target maximizing accuracy with fewer resources. Limiting the tree's depth can help prevent overfitting due to smaller datasets. LightGBM uses the histogram method. In the histogram method data is bucketed into bins using a histogram of the distribution instead

of finding the split points. The bin number can be handled by max_bin, min_data_in_bin, max_bin_my_feature, and bin_construct_sample_cnt. These bins are used to calculate, split the data, and calculate the gain. The dataset in LightGBM is sampled by Gradient-based One Side Sampling (GOSS). GOSS weights data points with a larger gradient higher when calculating the gain. By eliminating points with low prediction errors, GOSS lowers the sample size and enables attention to be paid to locations with high prediction errors. Figure 3.7 represents the architecture of LightGBM with SMOTE. The algorithm is trained using the synthesized database to get better performance.

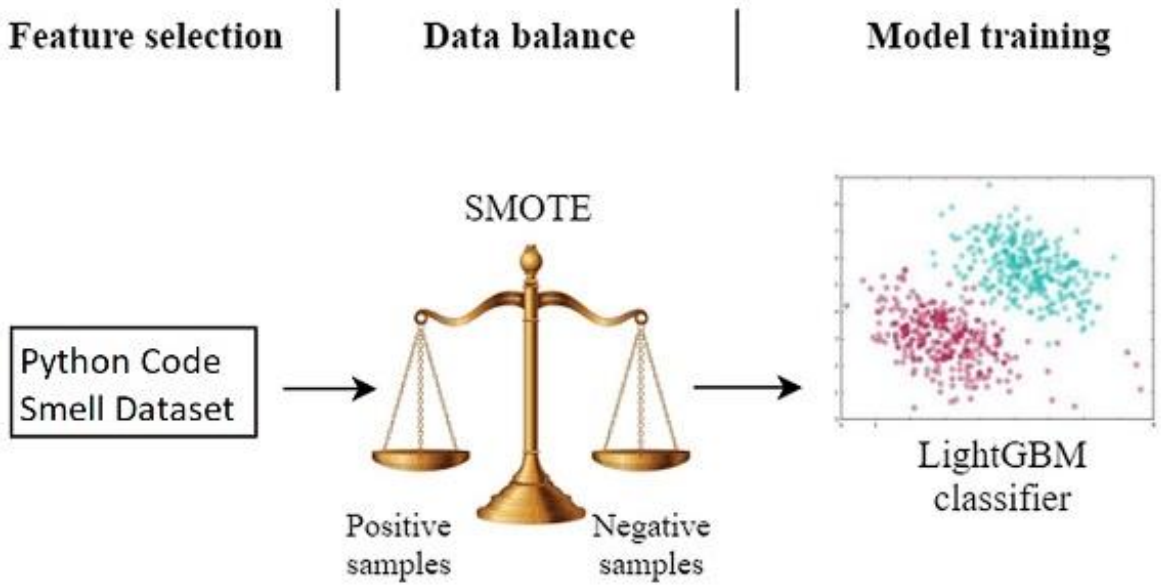

Figure 3.7: Architecture of LightGBM with SMOTE [19]

3.4.6 LightGBM with SMOTEENN

The SMOTEENN approach combines the ability of SMOTE to generate synthetic instances for minority classes with the capability of ENN to remove certain observations from both classes that are found to belong to a different class than the class of the observation and its K-nearest neighbor majority class. The SMOTE selects data from the minority class at random using this method. The distance between the random data and the K-nearest neighbors is then computed. The variations are multiplied by a random value ranging from 0 to 1. The multiplication result is then added to the minority class as a synthetic example. This method is continued until the necessary number of minority classes are created. Once the goal proportion

of the minority class is attained, ENN selects a number K as the number of nearest neighbors. ENN delivers the majority class from the K-nearest neighbor of the selected observation after determining the K-nearest neighbor of the selected observation from the remaining observations in the dataset. Both the observation and its K-nearest neighbor are removed from the dataset if their classes are different from the majority class of their neighbors. This technique is repeated until each class reaches the necessary percentage.

DTs produced by LightGBM develop leaf-by-leaf. Only one leaf may split under the specified conditions, depending on gain. Asymmetrical trees and Greddy growth are the two varieties of leaf-wise tree growth. Whereas asymmetrical trees aim to maximize accuracy with fewer resources, greedy growth concentrates on the most productive path. With smaller datasets, overfitting can be avoided by limiting the tree's depth. The histogram approach is used by LightGBM. Rather than identifying the split points, the histogram approach uses a histogram of the distribution to bucket data into bins. Bin_construct_sample_cnt, min_data_in_bin, max_bin, and max_bin_my_feature can all manage the amount of bins. The data is divided and the gain is computed using these bins. In LightGBM, the dataset is sampled using GOSS. When determining the gain, GOSS gives greater weight to data points with a steeper gradient. By eliminating points with low prediction errors, GOSS decreases the amount of samples and makes it possible to concentrate on points with large prediction errors. The LightGBM with SMOTEENN architecture is shown in Figure 3.8. To improve performance, the synthesized database is used to train the algorithm.

3.4.7 AdaBoost

AdaBoost or Adaptive Boosting is a ML model where multiple weak classifiers are combined and generates a perfect classifier that produces more accurate predictions. It learns from the mistakes of weak classifiers and then converts the stronger ones. AdaBoost, is also known as meta-learning is a sequential ensemble learning method. It assigns higher weights to misclassified data points in an iterative process. The total of the predictions from the weak classifiers is used to compute the final model. It can be used in different ML algorithms for better performance. AdaBoost can perform well on imbalanced datasets but is sensitive to noisy data.

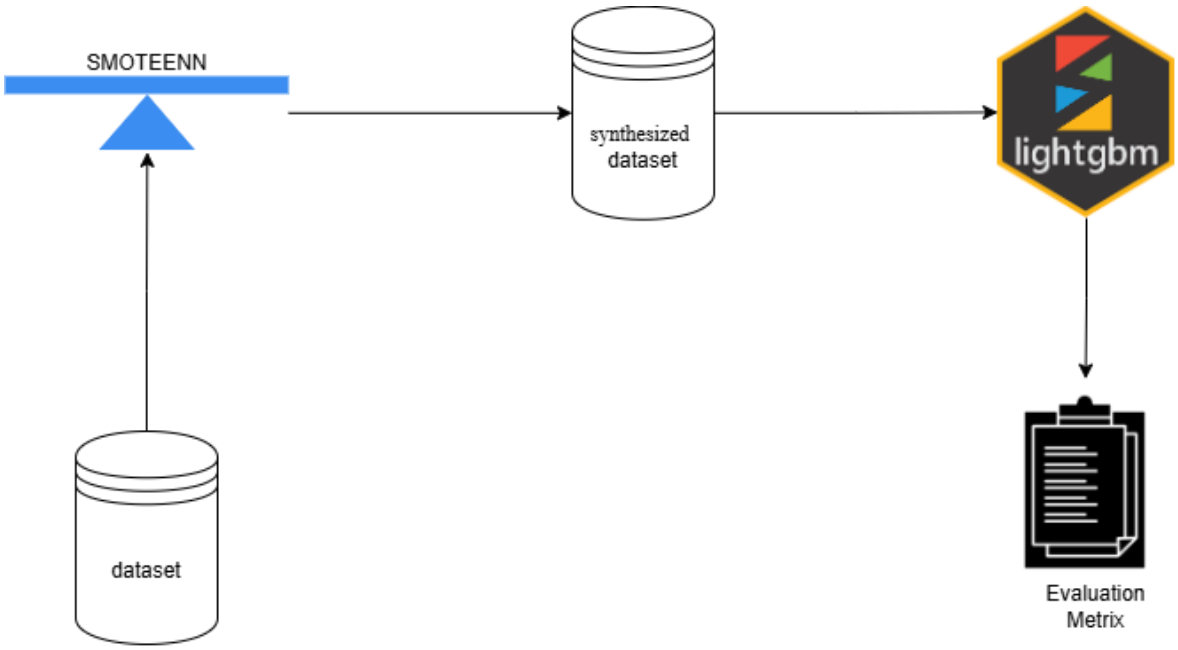

Figure 3.8: Architecture of LightGBM with SMOTEENN

AdaBoost chooses a weak classifier and trains it using the data based on their weight. Initially, all the data samples are given the same weight. It creates the classifiers for each variable to assess the correctness of classifiers sample to their target classes. The model identifies both correctly and incorrectly classified samples and assigns more weight to incorrectly classified samples. Additionally, the weight is determined by the classifiers' accuracy. The aim is to identify and classify the incorrectly classified sample in the next round. Until every data point has been accurately identified or the maximum number of iterations has been achieved, this procedure keeps going. The AdaBoost classifiers are displayed in Figure 3.9.

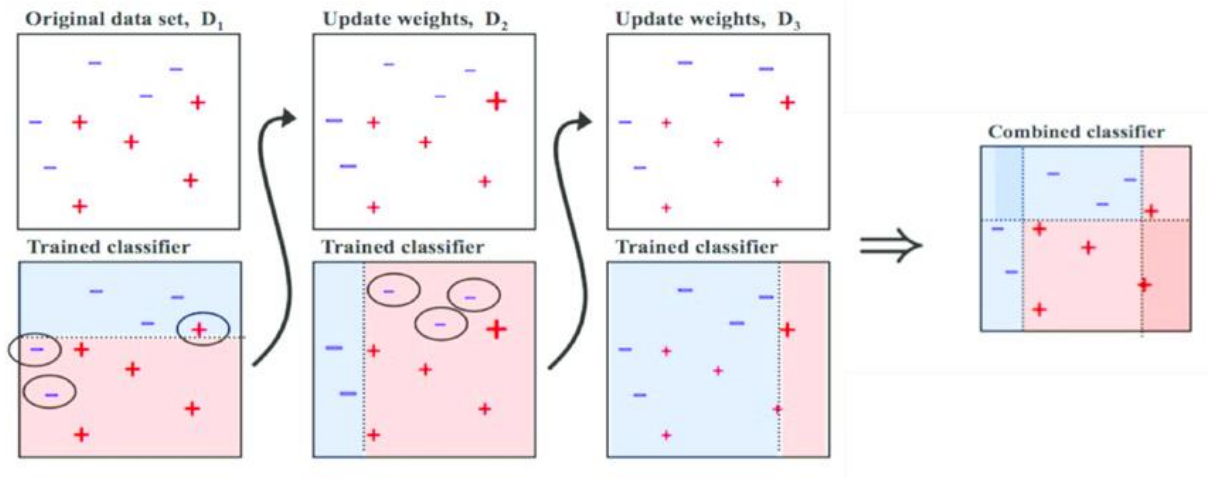

Figure 3.9: Classifiers of AdaBoost [20]

Let's dive into the mathematical part of AdaBoost.

$$x_i \epsilon R^n, y_i \epsilon \{-1, 1\} \quad (3.10)$$

Where,

- n = attributes in the dataset
- x = data points set
- y = targeted variable

Initially, the weights of all the samples are the same.

$$w = \frac{1}{N} \epsilon [0,1] \quad (3.11)$$

Here, N is the data point number. The formula for calculating the actual influence of the classifier is:

$$\alpha_t = \frac{1}{2} ln \frac{(1-TotalError)}{TotalError} \quad (3.12)$$

Total error is calculated by dividing the total number of misclassifications for a given training set by the training size. The value of alpha will be 0 when the classifiers classify half incorrectly and half correctly. Then the weights of the samples are updated using the following formula:

$$w_i = w_{i-1} * e^{\pm \alpha} \quad (3.13)$$

The value of alpha is positive if the sample was classified correctly and negative if the sample is misclassified. If the value of alpha is negative, the weight of the sample should be increased.

3.4.8 AdaBoost with SMOTE

Several ML algorithms use the SMOTE to handle imbalanced datasets and enhance algorithm performance. It first determines whether a class in the dataset is a minority. Next, The KNN technique is used by the SMOTE to identify the closest neighbors of the minority class. The chosen samples are connected to each of their closest neighbors by a straight line. Then, using a random scaling factor, a new point is added to each line. SMOTE keeps doing this until the required quantity of synthetic samples is produced. Then the AdaBoost algorithm can be trained using this synthetic dataset.

AdaBoost chooses a weak classifier and trains it using the data based on their weight. Initially, all the data samples are given the same weight. It creates the classifiers for each variable to assess the correctness of classifiers sample to their target classes. The model

identifies both correctly and incorrectly classified samples and assigns more weight to incorrectly classified samples. Additionally, the weight is determined by the classifiers' accuracy. The aim is to identify and classify the incorrectly classified sample in the next round. Until every data point is accurately categorized or the maximum iteration level is reached, this procedure keeps going. Figure 3.10 shows the architecture of AdaBoost with SMOTE. To improve performance, the synthesized database is used to train the algorithm.

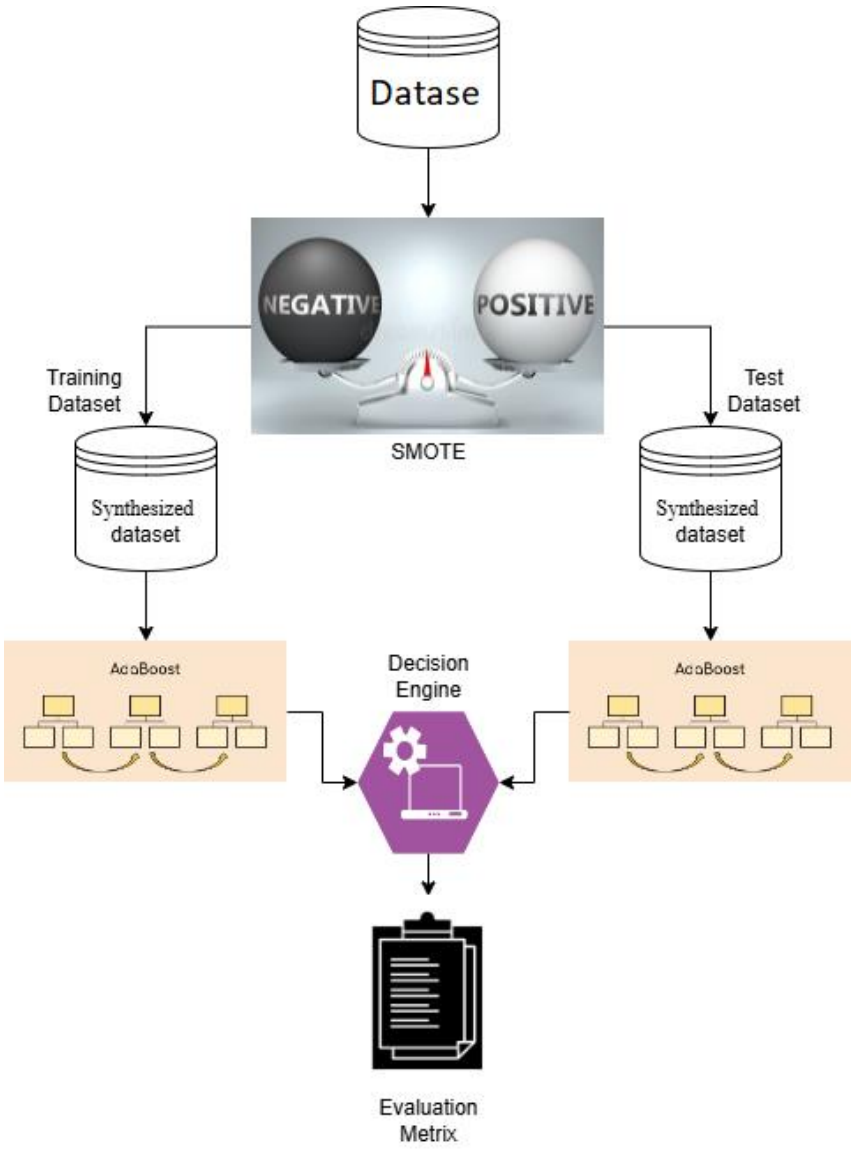

Figure 3.10: Architecture of AdaBoost with SMOTE

3.4.9 AdaBoost with SMOTEENN

SMOTE's capacity to create synthetic instances for minority classes and ENN's ability to exclude certain observations from both classes that are determined to belong to a different class than the observation's class and its K-nearest neighbor majority class are combined in the SMOTEENN approach. Using this technique, the SMOTE randomly chooses data from the minority class. Next, the distance between the K-nearest neighbors and the random data is calculated. The variants are given a random value between 0 and 1. Next, as a synthetic example, the multiplication result is added to the minority class. Until the required number of minority classes is produced, this process is repeated. Once the goal percentage of the minority class is attained, ENN selects a number K as the number of nearest neighbors. After identifying the K-nearest neighbor of the chosen observation from the remaining observations in the dataset, ENN provides the majority class from the K-nearest neighbor of the chosen observation. If the classes of the observation and its K-nearest neighbor disagree with the majority class of their neighbors, both are eliminated from the dataset. This technique is repeated until each class reaches the necessary percentage.

AdaBoost selects a weak classifier and uses the data to train it based on its weight. First, the same weight is assigned to each data sample. To determine whether the classifiers' samples are accurate for their target classes, it builds classifiers for every variable. Both correctly and erroneously classified samples are identified by the model, which gives the incorrectly classified samples additional weight. Additionally, the weight is determined by the classifiers' accuracy. In the following round, the goal is to locate and categorize the sample that was misclassified. This process continues until all of the data points have been correctly identified or the maximum number of iterations has been reached. The AdaBoost with SMOTEENN architecture is depicted in Figure 3.11. The algorithm is trained on the synthesized database to enhance performance.

3.4.10 Decision Tree with SMOTE

DT is a most popular and powerful ML algorithm. It is used to make decisions or predictions. It has a flowchart-like structure that consists of nodes and branches. The dataset is split into subsets based on Gini impurity, information gain, or entropy. The splitting continues until a stopping criterion is satisfied. If a new instance were randomly classified using the class distribution of the dataset, the Gini Impurity measures the likelihood that it would be erroneously classified.

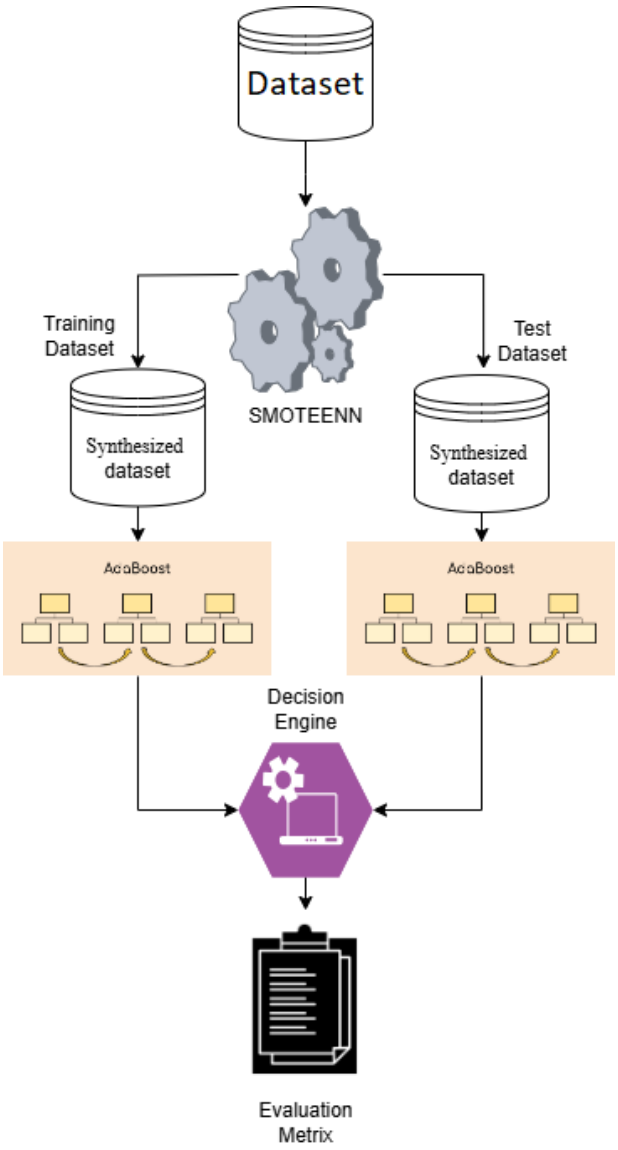

Figure 3.11: Architecture of AdaBoost with SMOTEENN

$$\text{Gini} = 1 - \sum_{i=1}^{n}(p_i)^2 \quad (3.14)$$

Here the probability of an instance will belong to a particular class is denoted by p_i. The dataset's level of impurity or uncertainty is measured by entropy.

$$\text{Entropy} = -\sum_{i=1}^{n} p_i log_2(p_i) \quad (3.15)$$

Here the probability of an instance will belong to a particular class is denoted by p_i. Information Gain quantifies the decrease in Gini impurity or entropy following an attribute-based dataset split.

$$\text{Information Gain} = Entropy_{parent} - \sum_{i=1}^{n}(\frac{|D_i|}{|D|} * Entropy(D_i)) \quad (3.16)$$

Here D_i is the subset of D after splitting. Pruning is a technique used in DT to overcome overfitting. By eliminating nodes that have minimal ability to identify instances, pruning reduces the tree's size. Figure 3.12 represents the architecture of a DT.

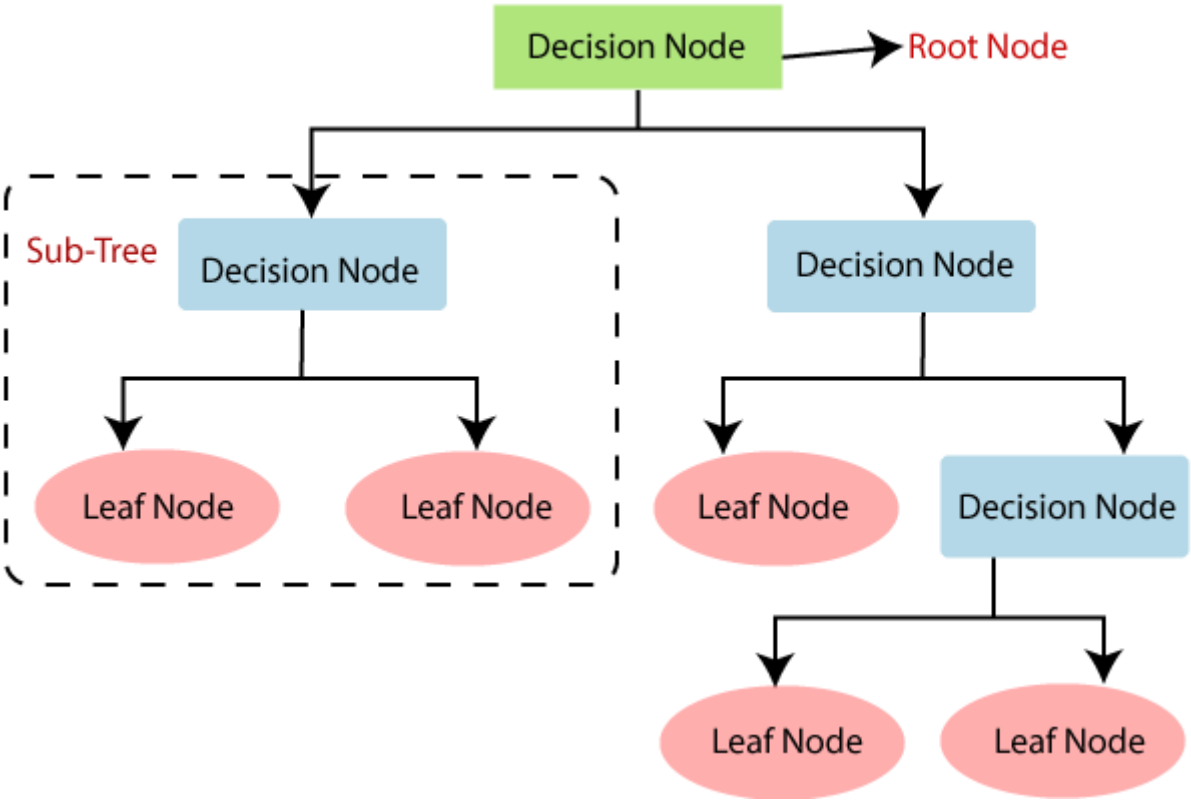

Figure 3.12: Architecture of Decision Tree [21]

The SMOTE is used by several ML algorithms to improve algorithm performance and manage imbalanced datasets. It first ascertains if a dataset class is a minority. After choosing the minority class, the SMOTE uses the KNN approach to determine the class's closest neighbors. A straight line connects the selected samples to each of their nearest neighbors. Next, a new point is added to each line using a random scaling factor. This is how SMOTE keeps going until the required quantity of synthetic samples is produced. The DT algorithm can then be trained using this synthesized dataset. The DT construction using SMOTE is shown in Figure 3.13.

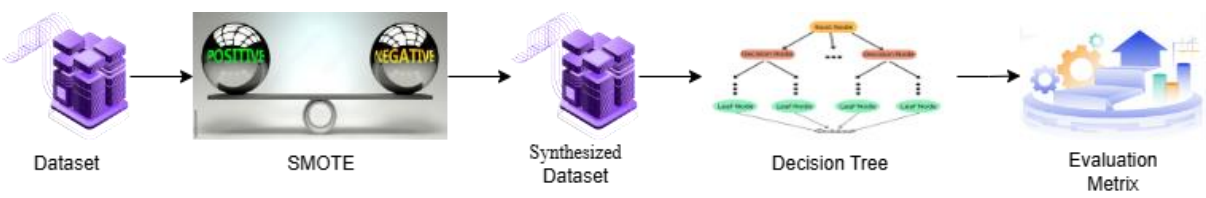

Figure 3.13: Structure of Decision Tree with SMOTE

3.4.11 Random Forest with SMOTE

A group of several DTs called RF cooperate to produce a single, more accurate outcome. It is also known as Random Decision Tree (RDT). It is a very powerful ML algorithm. RF generates many DTs throughout the training stage. A random portion of the dataset in each partition is used by each tree to measure a random subset of features. This technique improves the prediction performance and reduces the risk of overfitting. The final results are aggregated from all the trees by the algorithm. It uses averaging or voting to aggregate the results.

RF selects K data points from the dataset randomly. Then it builds the DTs associated with these selected data points. For a new data point, it finds the prediction of each DT. These trees operate independently. The RF equation is represented as,

$$g(x) = f_0(x) + f_1(x) + f_2(x) + \dots f_n(x) \quad (3.17)$$

Here,

- g = final model
- f_i = a simple base model

The predictor is represented as,

$$\tilde{f}_{B.r.f}(x) = \frac{1}{B} \sum_{b=1}^{B} T(x; \Theta_b) \quad (3.18)$$

The final feature importance is:

$$RFfi_{sub(i)} = \frac{\sum_{i=1}^{T} fi_{sub(i)}}{T} \quad (3.19)$$

Figure 3.14 shows the architecture of RF.

Several ML algorithms use the SMOTE to handle imbalanced datasets and enhance algorithm performance. It first determines whether a class in the dataset is a minority. Next, The KNN technique is used by the SMOTE to identify the closest neighbors of the minority class. The chosen samples are connected to each of their closest neighbors by a straight line. Then, using a random scaling factor, a new point is added to each line. SMOTE keeps doing this until the required quantity of synthetic samples is produced. Then the RF algorithm can be trained using this synthetic dataset. In Figure 3.15, the architecture of Random Forest with SMOTE is given.

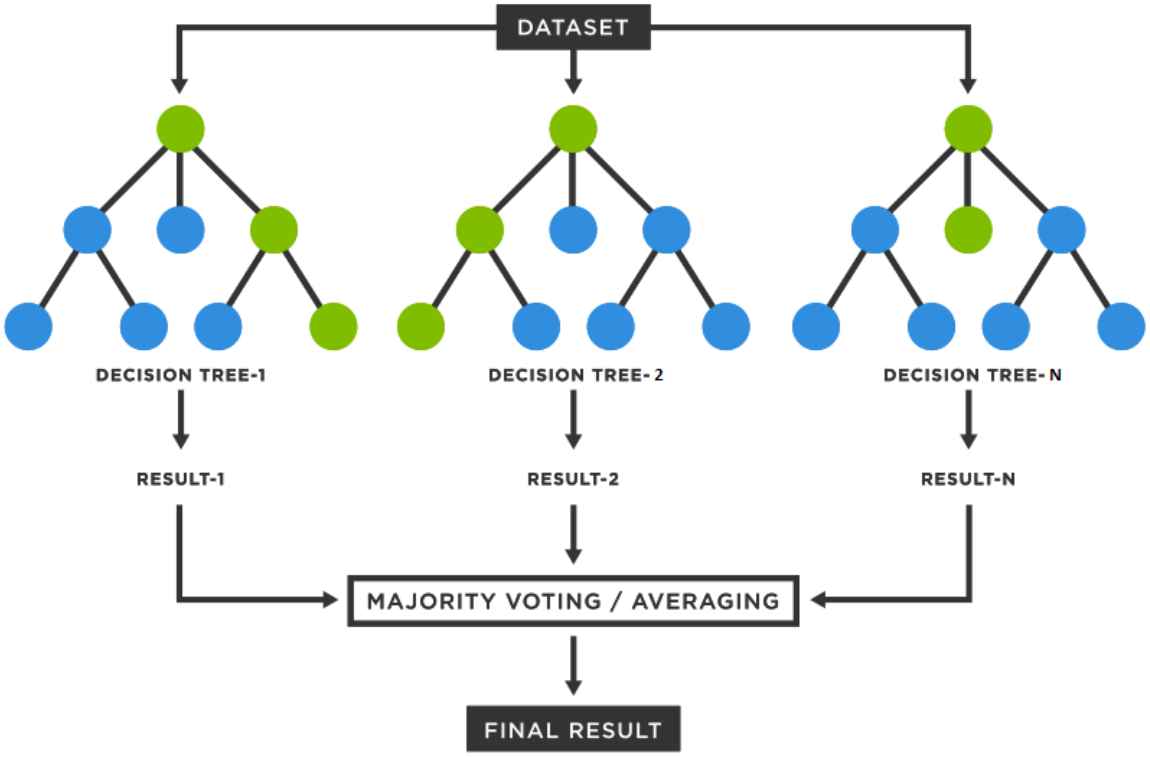

Figure 3.14: Architecture of Random Forest [22]

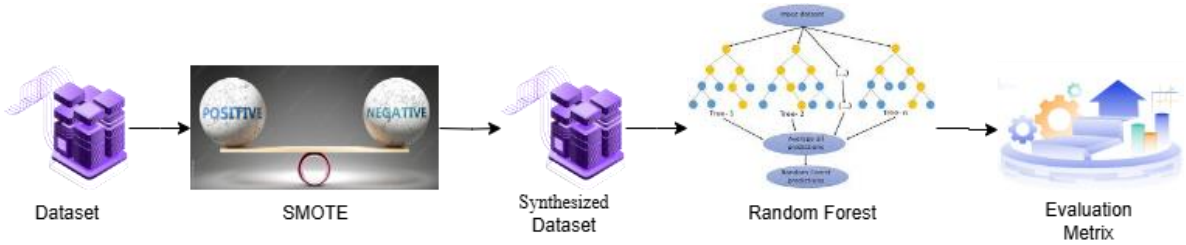

Figure 3.15: Architecture of Random Forest with SMOTE

3.5 Models Execution Process

This section discusses the execution process of all models. We discuss data preprocessing, feature scaling, feature selection approach, dataset import, optimization, resampling, splitting the dataset, training, and evaluation in this section. Figure 3.16 represents our proposed execution process.

3.5.1 Import Dataset

Initialized the chosen model with desired datasets to speed up training and improve convergence. The Comma-Separated Values (CSV) file is mounted to the model.

3.5.2 Optimization

Pandas is used to load the dataset from a CSV file. The collection includes labels that match both smelly and non-smelly portions of the source code. In this step, we can make sure that the data will be available for future processing including feature scaling, feature selection, resampling, dataset splitting into training and test datasets, and perform additional preprocessing for training the model.

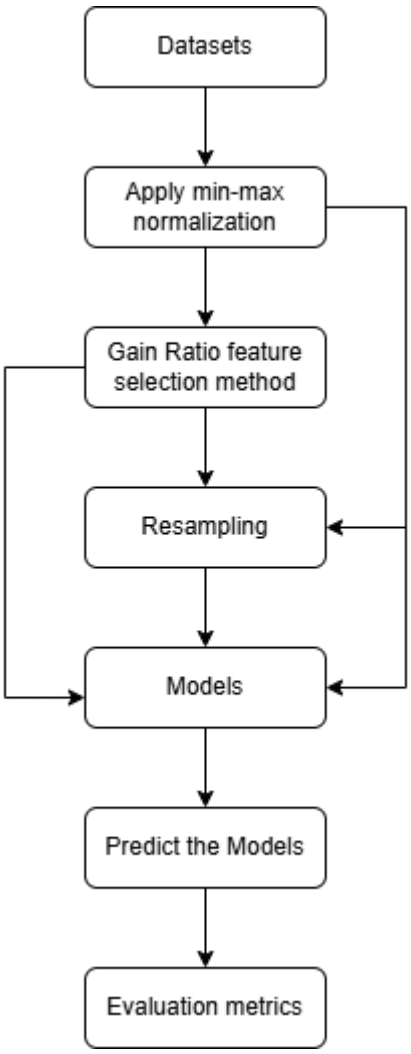

Figure 3.16: Proposed models execution process

3.5.3 Feature Scaling

The majority of the time, datasets have varying feature ranges. Those datasets are not suitable for direct application of certain ML and classification methods. One technique for preparing data is feature scaling. Normalizing the range of data features is its goal. One popular

preprocessing method used before ML training is feature scaling. It enhances both the performance of the models and the quality of the data. It also covers the dataset's many ranges. On the feature-scaled dataset, several ML methods can converge more quickly.

For feature scaling in this investigation, we used the min-max normalization (MMN) technique. The feature values are converted between 0 and 1 by MMN. In this case, 0 will be the lowest value and 1 will be the highest. The remaining numbers will range from 0 to 1. Using the following formula, the min-max normalization rescales and adjusts a feature's values:

$$X_{scaled} = \frac{X - X_{min}}{X_{max} - X_{min}} \quad (3.20)$$

Where,

- X = Original value
- X_{scaled} = Normalized value between 0 to 1
- X_{max} = Max value of X
- X_{min} = Min value of X

3.5.4 Feature Selection

The goal of feature selection, a data preprocessing method, is to identify the most significant characteristics in the dataset. By removing qualities that are not significant, it reduces the number of characteristics in the dataset. Performance and dependability can be improved by applying the feature selection technique. In this investigation, we selected features using the Gain Ratio method. The feature will be given a gain score ranging from 0 to 1 using this method. If a feature's score is lower than the mean gain score, it will be eliminated from the feature collection. They are thought to be inconsequential traits. Features will be kept if their scores are at least as high as the mean gain score.

Both the Large Class and Long approach datasets have fewer features after using the gain ratio feature selection approach. There were 12 features rather than 18 in the Large Class dataset. The following features were chosen: volume, calculated program length, effort, difficulty, number of single comments, time required to program, number of delivered bugs, number of logical lines of code (LLOC), number of lines of code (LOC), number of comments, number of blank lines, and number of source lines of code (SLOC). There were eight features rather than eighteen in the Long Method dataset. Source lines of code (SLOC), logical lines of code (LLOC), volume, computed program length, effort, difficulty, number of delivered bugs, and programming time are the features that were chosen. Table 3.2 presents the selected feature for each dataset. Figure 3.17 shows the gain ratio for every feature that was chosen.

3.5.5 Resampling

Some datasets may contain a minority number of data samples of any specific classes. Training ML algorithms with these datasets may lead to poor performance and generate less accurate predictions. To overcome this problem, we use the resampling technique in this study to improve the performance of models and produce more accurate predictions.

Resampling is a series of techniques to generate more synthetic samples based on any specific samples. It helps to improve the overall accuracy. It also helps to balance an imbalanced dataset by generating synthetic samples of the minority classes in a given dataset. In our study, we use the two most popular and advanced resampling techniques called SMOTE and SMOTEENN.

Dataset	**Large Class**	**Long Method**
Number of Features	12	8
Selected Features	number of logical lines of code (LLOC), number of lines of code (LOC), number of comments, number of source lines of code (SLOC), calculated program length, number of blank lines, difficulty, volume, time required to program, effort, number of delivered bugs, number of single comments	number of source lines of code (SLOC), number of logical lines of code (LLOC), volume, calculated program length, effort, difficulty, number of delivered bugs, time required to program

Table 3.2: Selected features for both datasets

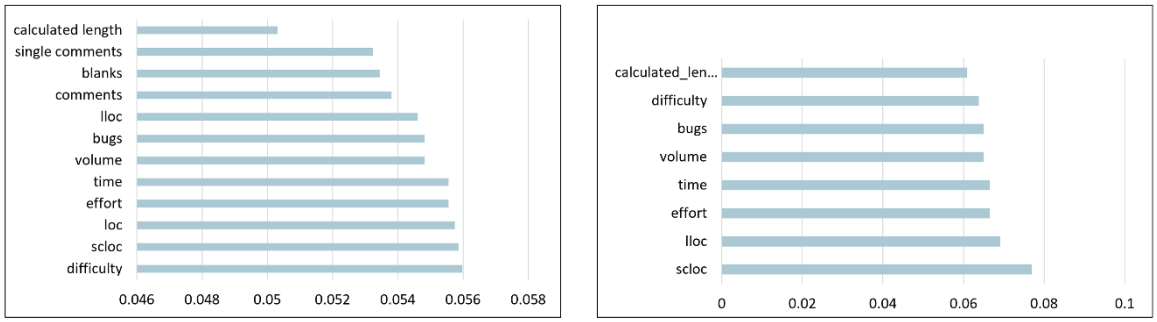

Figure 3.17: The gain ratio for every feature that was chosen (a) Selected features from a large class dataset (b) Features are chosen from the long method dataset [1]

3.5.6 Split the Dataset

A training set and a testing set are the two subsets of the dataset that are now separated. This is accomplished by using the train test split function from Skit-Learn. The test set is used for evaluation to determine the model's performance, whereas the training set is used for training.

3.5.7 Load the Models

In this phase, we import our desired models from their predefined library. Once the models are imported, they are ready for training and testing. These components play very important roles in both training and carrying out the processing tasks.

3.5.8 Model Validation

Using stratified 10-fold cross-validation, the models were verified. The dataset is randomly split into ten comparable subsets during this process. Ten iterations of the training and testing procedure are conducted. The models are trained using nine folds each time, and they are tested using the remaining one. For testing, every fold is used precisely once. After the procedure is repeated ten times, the final result is computed by averaging the outcomes of the iterations. To offer more dependable performance, this procedure can solve overfitting and produce more accurate outcomes [23].

3.5.9 Evaluation

We assess our trained model's performance using training data in this “Evaluation” section. As we go through the test dataset, we feed the test data as input so that the models can

make predictions. We calculate two evaluation metrics, accuracy and MCC score. These measurements shed light on how well the model predicts the smell of Python code. By analyzing the evaluation results, we were able to fully comprehend the model's functionality, which enabled us to assess how effectively it anticipates code smells in real-world scenarios.

3.6 Evaluation Metrics

Python code smell detection is examined in this work as a binary classification problem. The classification of code samples into smelly and non-smelly is the goal of our models. We employed Accuracy, and MCC scores to assess the models.

3.6.1 Accuracy

One of the profuse used performance metrics for classification issues is accuracy. It is also applied to categorization challenges with code smells. The proportion of accurately predicted tuples to all tuples is known as accuracy. The number of accurately predicted points divided by the total number of points in the dataset is one way to express it.

$$\text{Accuracy} = \frac{TP+TN}{TP+FP+TN+FN} \times 100 \quad (3.21)$$

Here,

- TP = True Positive
- FP = False Positive
- TN = True Negative
- FN = False Negative

The higher accuracy indicates the better performance of the model.

3.6.2 Matthews Correlation Coefficient (MCC)

MCC is a statistical measure that may be trusted. When the model predictions function well, it generates a higher score. There are the four confusion matrix classifications that are used. They are: True positive, false positive, true negative, and false negative. MCC has a value between +1 and -1. A perfect misclassification is represented by -1, a perfect model by +1, and a random prediction by 0. MCC is regarded as the best metric for unbalanced datasets. Each class performs well when most of its members are accurately classified. For the categorization of imbalanced datasets, MCC is therefore advised more than F1-score and other metrics [24]

[25]. To obtain more precise and consistent model identification performance throughout the unbalanced Python code smell datasets, we employed MCC as an assessment metric.

$$\text{MCC} = \frac{TP \times TN - FP \times FN}{\sqrt{(TP+FP)(TP+FN)(TN+FP)(TN+FN)}} \quad (3.22)$$

The model performs at its peak when the MCC rate is higher.

Chapter 4

Results and Discussions

This section contains our findings as well as an analysis of the outcomes of every model we utilized in the experiment. The experiment's outcomes and the 11 ML techniques are displayed. We used Google Colab and built-in Python Libraries to execute the models.

4.1 Performance Comparison

We used ensemble learning baselines to detect Python Large Class and Long Method code smells, and the results (accuracy and MCC score) are shown in Table 4.1. In both Large Class and Long Method code smell detection, the LightGBM with SMOTEENN model performed the best. In the Long Method code smell detection, the XGBoost with the SMOTEENN model also obtained the same accuracy and MCC score. The XGBoost and SMOTEENN models performed the second best overall. Without resampling XGBoost, LightGBM, and AdaBoost provide less accurate performance. Both SMOTE and SMOTEENN improve the performance of each model. SMOTEENN leads to provide the highest accuracy and MCC score across both Large Class and Large Method code smell. XGBoost performs comparably well without resampling than LightGBM and AdaBoost.

XGBoost and LightGBM both achieved around 92-95% accuracy and 0.77-0.89 MCC score without any resampling. However, AdaBoost performs slightly lower than XGBoost and LightGBM. It achieves 91.41% accuracy with the MCC score of 0.73 for Large Class and 95.53% accuracy with the MCC score of 0.88 for Long Method. The performance of each model is improved by applying SMOTE to handle the imbalance class. XGBoost's accuracy improves to 97.27% for Long Method and LightGBM improves the accuracy to 94.66% for Large Class code smell. With SMOTEENN the performance reached the highest accuracy and MCC score. XGBoost reaches 98.33% accuracy and 0.97 MCC score for Large Class and 99.79 accuracy with MCC of 1.00 for Long Method. On the other hand, LightGBM reaches 98.47% accuracy and 0.97 MCC score for Large Class and 99.79 accuracy with MCC of 1.00 for Long Method. AdaBoost also greatly benefited from SMOTEENN. In addition, DT shows 91.53% accuracy and RF 93.91% accuracy for Large Class with SMOTE while achieving 96.50% accuracy and 97.19% accuracy for Long Method respectively. The comparative accuracy and

MCC Scores for identifying both kinds of smells across all models are displayed in Figures 4.1 and 4.2, respectively.

Models	Large Class		Long Method	
	Accuracy	MCC	Accuracy	MCC
XGBoost	92.65	0.77	95.84	0.89
XGBoost with SMOTE	94.43	0.89	97.27	0.95
XGBoost with SMOTEENN	98.33	0.97	99.79	1.00
LightGBM	92.65	0.77	95.68	0.88
LightGBM with SMOTE	94.66	0.89	97.07	0.94
LightGBM with SMOTEENN	98.47	0.97	99.79	1.00
AdaBoost	91.41	0.73	95.53	0.88
AdaBoost with SMOTE	91.70	0.84	96.76	0.94
AdaBoost with SMOTEENN	97.35	0.95	99.67	0.99
Decision Tree with SMOTE	91.53	0.83	96.50	0.93
Random Forest with SMOTE	93.91	0.88	97.19	0.94

Table 4.1: Performance outcomes for detecting code smells using the large class and long method

4.2 Impact of Feature Selection

To enhance the models' performance, we concentrated on the feature selection approach in this study. Selecting a subset of the original features is known as feature selection. This research, we employ the Gain Ratio feature selection method. The high-dimensional dataset's properties are ranked by the gain ratio. Based on information gain and number of a single

feature's outcomes, Gain Ration selects the best features. The accuracy and MCC Score of every algorithm are compared before and after the feature selection technique is applied in Table 4.2. The result indicates that the XGBoost, Decision Tree with SMOTE, and Random Forest with SMOTE perform better in Large Class code smell detection when using all the features and the feature selection method provides better performance for Long Method code smell detection. LightGBM performs better using all the features while the dataset is resampled. AdaBoost provides better performance using all the features when the dataset hasn't been resampled. Figures 4.3, 4.4, 4.5, and 4.6 present the Comparative Accuracy and MCC score of Large Class and Long Method using the Feature Selection method and All Features.

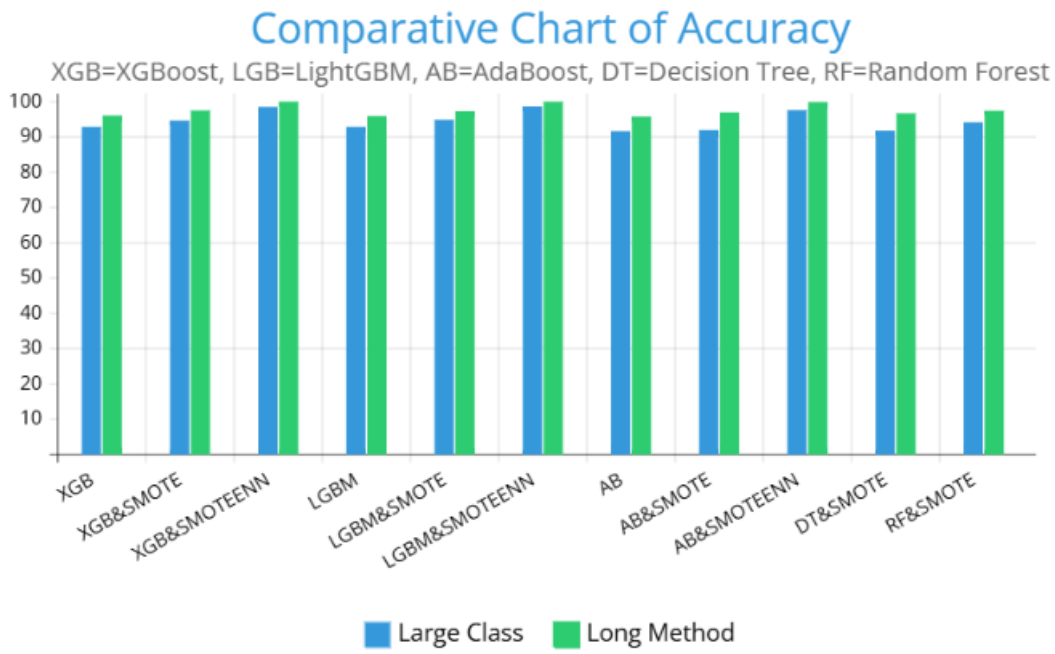

Figure 4.1: The Bar Chart Shows Accuracy of the Models

4.3 Comparison of Our Results with Related Works

Our results are briefly compared with those of Sandouka and Aljamaan [1] in Figure 4.7 and Table 4.3. Using the Python code smell dataset provided by Sandouka and Aljamaan, these techniques employed ML techniques. In the Large Class dataset, our LightGBM with SMOTEENN approach achieved 98.47% accuracy with an MCC score of 0.97 while Sandouka and Aljamaan got the highest accuracy of 92.7% with an MCC score of 0.77 for the RF model. In the Long Method dataset, Sandouka and Aljamaan represented the best performance of the DT with 95.9% accuracy and 0.90 MCC score. Additionally, in our proposed method provides

a notable improvement in the performance also in the given dataset. Both XGBoost with SMOTEENN and LightGBM with SMOTEENN achieve 99.79% accuracy with a 1.00 MCC score. This comparison indicates that we achieved an outstanding improvement in performance compared to the previous work.

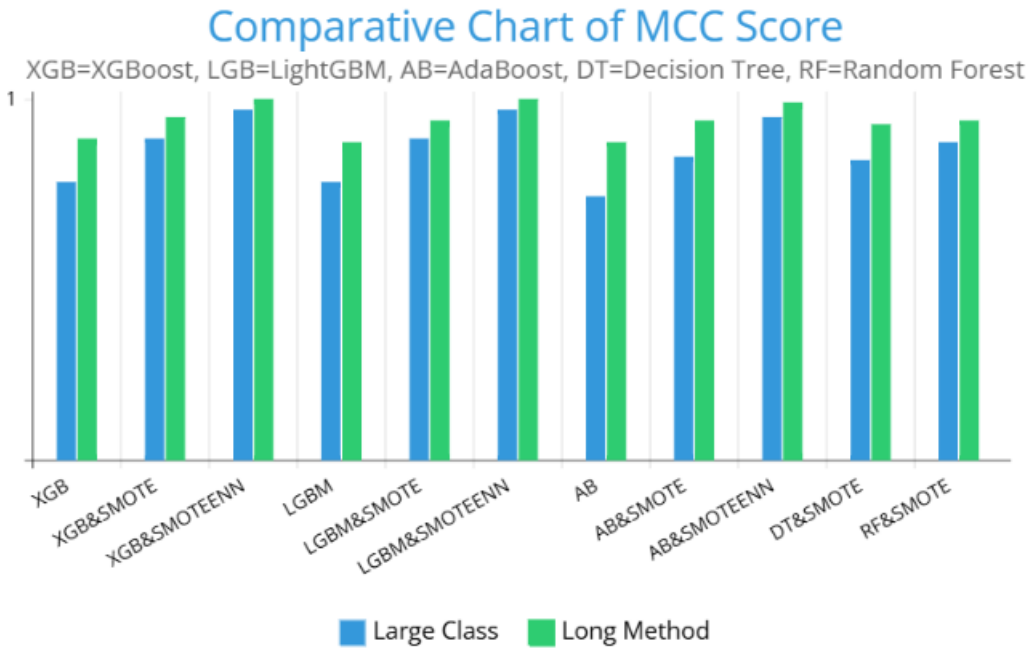

Figure 4.2: The Bar Chart Shows the MCC Score of the Models

We also show that RF and DT perform better when we use the SMOTE resampling technique with them. Random Forest with SMOTE achieved an accuracy of 93.91% with a 0.88 MCC score for Large Class code smell detection while Decision Tree with SMOTE achieved 96.50% accuracy with a MCC score of 0.93 for Long Method code smell detection. It is also presented that the RF algorithm with the SMOTE resampling technique provides 97.19% accuracy with a 0.94 MCC score for Long Method code smell detection with selected features and 94.92% accuracy with a 0.90 MCC score for Large Class code smell detection using all the features.

Code Smell Dataset	Models	Gain Ratio for Feature Selection		All Features	
		Accuracy	MCC	Accuracy	MCC
Large Class	XGBoost	92.65	0.77	93.12	0.78
	XGBoost with SMOTE	94.43	0.89	95.78	0.92
	XGBoost with SMOTEENN	98.33	0.97	98.60	0.97
	LightGBM	92.65	0.77	92.62	0.77
	LightGBM with SMOTE	94.66	0.89	95.96	0.92
	LightGBM with SMOTEENN	98.47	0.97	98.76	0.98
	AdaBoost	91.41	0.73	91.94	0.74
	AdaBoost with SMOTE	91.70	0.84	93.16	0.86
	AdaBoost with SMOTEENN	97.35	0.95	98.36	0.97
	Decision Tree with SMOTE	91.53	0.83	92.04	0.84
	Random Forest with SMOTE	93.91	0.88	94.92	0.90
Long Method	XGBoost	95.84	0.89	95.44	0.88
	XGBoost with SMOTE	97.27	0.95	97.57	0.95
	XGBoost with SMOTEENN	99.79	1.00	99.23	0.98
	LightGBM	95.68	0.88	95.55	0.88
	LightGBM with SMOTE	97.07	0.94	97.45	0.95
	LightGBM with SMOTEENN	99.79	1.00	99.15	0.98
	AdaBoost	95.53	0.88	95.56	0.88
	AdaBoost with SMOTE	96.76	0.94	96.91	0.94
	AdaBoost with SMOTEENN	99.67	0.99	99.07	0.98
	Decision Tree with SMOTE	96.50	0.93	95.67	0.92
	Random Forest with SMOTE	97.19	0.94	97.08	0.94

Table 4.2: Feature selection techniques' impact on prediction models

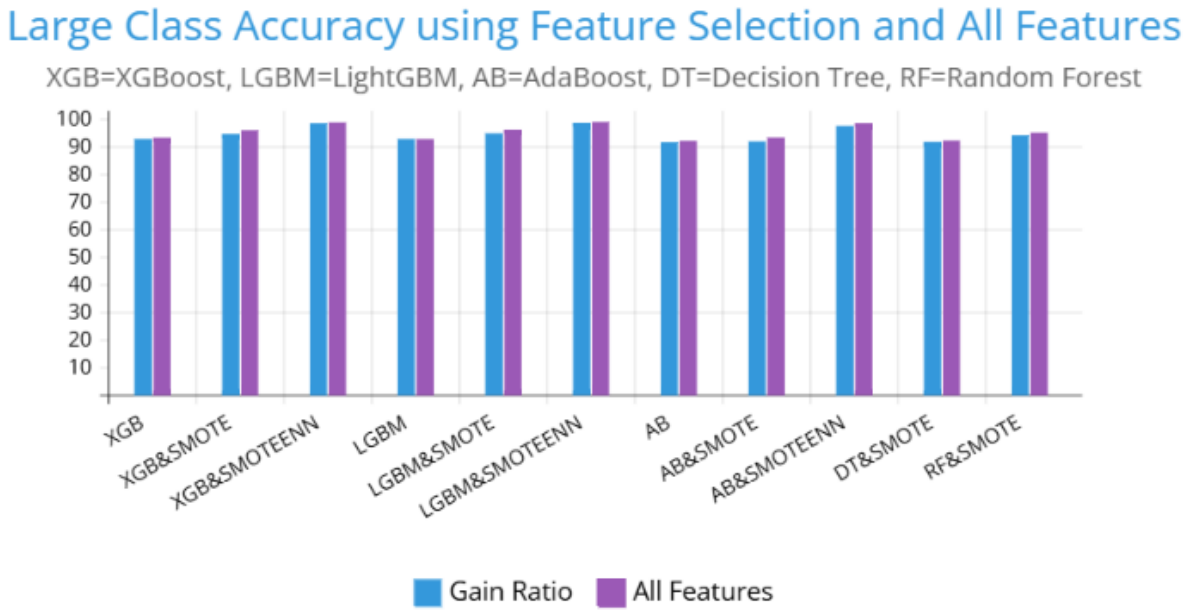

Figure 4.3: Comparative Accuracy of Large Class using Feature Selection method and All Features

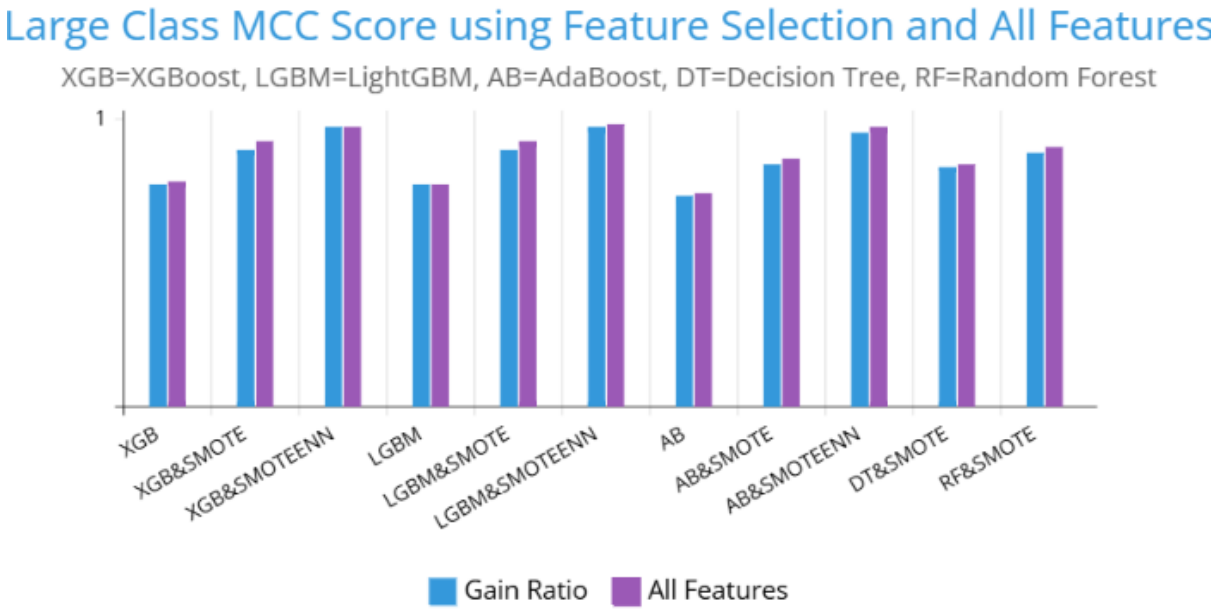

Figure 4.4: Comparative MCC Score of Large Class using Feature Selection method and All Features

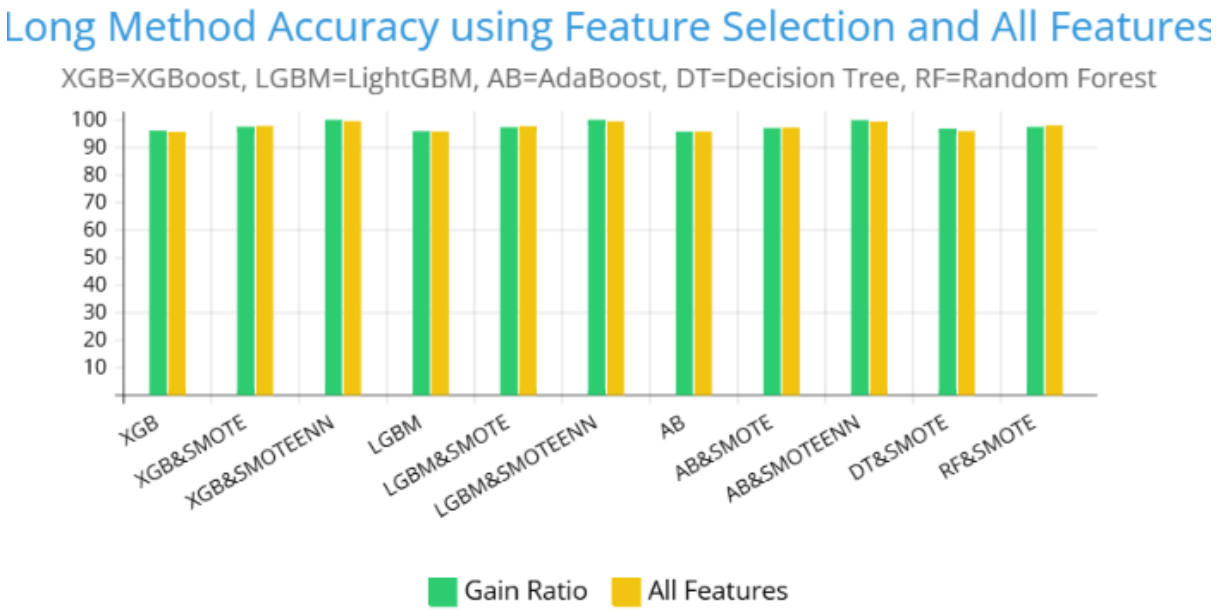

Figure 4.5: Comparative Accuracy of Long Method using Feature Selection Method and All Features

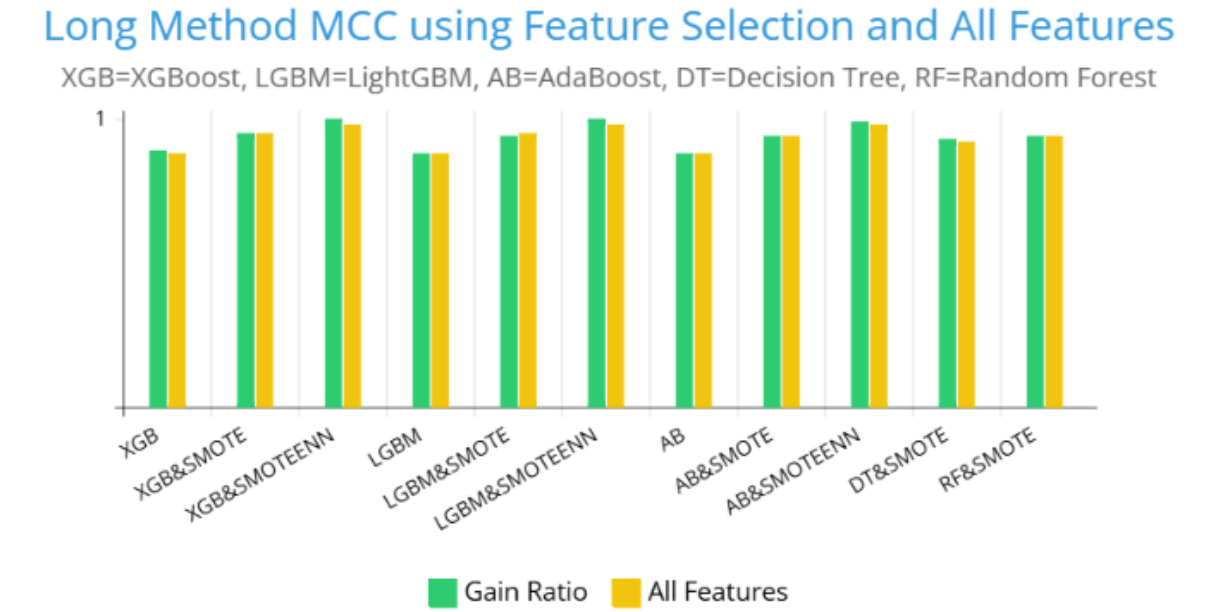

Figure 4.6: Comparative MCC Score of Long Method using Feature Selection method and All Features

Code Smell	Sandouka and Aljamaan			Proposed Method		
	Best Models	Highest Accuracy	Highest MCC	Best Models	Highest Accuracy	Highest MCC
Large Class	Random Forest	92.7	0.77	LightGBM with SMOTEENN	98.47	0.97
Long Method	Decision Tree	95.9	0.90	XGBoost with SMOTEENN and LightGBM with SMOTEENN	99.79	1.00

Table 4.3: Comparison of our results with Sandouka and Aljamaan

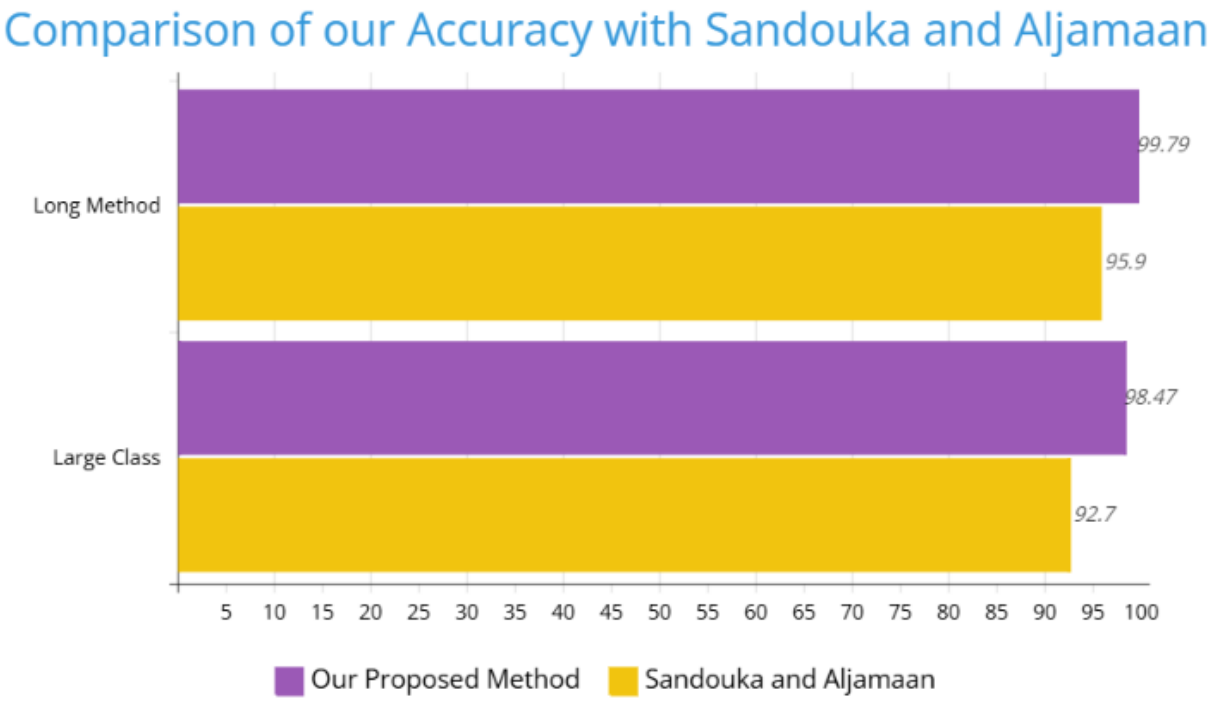

Figure 4.7: Comparison of our Accuracy with Sandouka and Aljamaan

Chapter 5

Conclusion and Future Directions

5.1 Interpretation of Results

In evaluating the prediction-ability of Python Large Class code smell detection, which comprises two classes – smelly and non-smelly, we observed notable variations in accuracy across different models. Firstly, using traditional ML models with SMOTE, the DT with SMOTE achieved an accuracy of 91.53% and the RF with 93.91%.

Subsequently, when examining various ensemble learning models including XGBoost, LightGBM, and AdaBoost, we found considerable diversity in accuracy scores. Notably, LightGBM with SMOTEENN achieved the highest accuracy at 98.47, closely followed by XGBoost with SMOTEENN (98.33%), and AdaBoost with SMOTEENN (97.35%). LightGBM with SMOTE and XGBoost with SMOTE also demonstrated competitive performance, achieving accuracies of 94.66% and 94.43% respectively.

Comparing the accuracy difference between ML and ensemble learning models, LightGBM with SMOTEENN outperformed Random Forest with SMOTE by 4.56%. Overall, these findings underscore the effectiveness of ensemble learning models, particularly LightGBM and XGBoost, in achieving high accuracy in Python Large Class code smell detection compared to traditional ML models like DT and RF.

In evaluating the performance of Python Long Method code smell detection, which comprises two classes – smelly and non-smelly, we observed notable variations in accuracy across different models. Firstly, using traditional ML models with SMOTE, the DT with SMOTE achieved an accuracy of 96.50% and the RF with 97.19%.

Subsequently, when examining various ensemble learning models including XGBoost, LightGBM, and AdaBoost, we found considerable diversity in accuracy scores. Notably, LightGBM with SMOTEENN and XGBoost with SMOTEENN achieved the highest accuracy at 99.79, closely followed by AdaBoost with SMOTEENN (99.67%). LightGBM with SMOTE and XGBoost with SMOTE also demonstrated competitive performance, achieving accuracies of 97.07% and 97.27% respectively.

Comparing the accuracy difference between ML and ensemble learning models, XGBoost with SMOTEENN outperformed Random Forest with SMOTE by 2.60%. Overall, these findings underscore the effectiveness of ensemble learning models, particularly LightGBM and XGBoost, in achieving high accuracy in Python Long Method code smell detection compared to traditional ML models like DT and RF.

For both types of code smells, LightGBM with SMOTEENN achieved outstanding performance that is better than similar existing research. XGBoost has taken the second position. While using ensemble learning, SMOTE, and SMOTEENN enforce toward the better accuracy. AdaBoost provides less performance compared to other ensemble learning.

5.2 Strengths and Limitations

This research on Python code smell detection using ensemble learning and resampling techniques has notable strengths and some limitations. The study achieved high accuracy, particularly with LightGBM and SMOTEENN, which performed exceptionally well at detecting code smells like Large Class and Long Method. By using SMOTE and SMOTEENN, the research effectively addressed the challenge of imbalanced datasets. Eleven distinct models were also assessed in the study, ranging from ensemble techniques like XGBoost, LightGBM, and AdaBoost to more conventional ML models like DT and RF. Further, the comparison of this feature selection to the full feature analysis gives some understanding of which features are more significant so that we can use less computational resources without decreasing the precision.

However, there are limitations: This study is bound to Python and, hence may not be generalized to other languages. Ensemble models like SMOTEENN are also complex, which can make them hard to interpret. This study considered only two datasets, which may not generalize predictions for all types of code smells. Finally, while SMOTE and SMOTEENN address class imbalance, other advanced techniques could be explored.

5.3 Contributions and Novelty

This research makes several unique contributions to Python code smell detection by using advanced resampling techniques and ensemble learning models. It shows that combining ensemble methods like XGBoost, LightGBM, and AdaBoost with resampling methods like SMOTE and SMOTEENN can greatly improve accuracy. The study gives useful information

that can help in handling instances of smelly code, which is a rare occurrence. It also differs due to the comparison of eleven models, which are related to both traditional ML and ensemble models. This paper also explains the effect of feature selection. This research is one of the few to focus specifically on Python. It outperforms previous research on similar tasks, setting a higher benchmark for accuracy in Python code smell detection. Finally, taken together, this work provides a useful and new contribution to software quality assessment.

5.4 Generalization and Applicability

The methods tested in this research, particularly the combination of ensemble models with resampling techniques, achieved high accuracy and could be applied beyond Python to other programming languages. This research also demonstrates an approach that could be valuable in other contexts where smelly code instances are rare by addressing the common issue of imbalanced datasets.

However, the findings are based on two specific datasets—targeting "Large Class" and "Long Method" code smells. Applying these methods to other code smell types may require further customization and validation. It is also useful to think of Python as a programming language on top of which you can build your ideas. However, the concepts underlying this work do not apply very well to Java or C++. Planning can be done to use the results of any further trends for other application areas both by cross-language crossing and to create a set of model translations that can be used in different programming languages.

It also discusses the feature selection aspect by saying that using all the features is not only a horrible disservice to the high accuracy but is also a use of computational resources that may not be available where the study will be done. Such models are more reasonable to try within similar contexts while reporting the results within these models reported here regarding data and feature selection might be useful for fine-tuning and practical use of such models in real-life environments of software development, testing, and code validation.

5.5 Future Research Directions

Several promising directions exist for future research on Python code smell detection using ensemble learning and advanced resampling techniques. Generalized method across different programming languages and cross-language code smell detection for synthesizing universal models of software quality assessment. In addition, searching for bigger kinds of code

smells like ‘feature envy’ or ‘duplicated code’ would make detection tools more comprehensive. In building confidence in the predictions, model interpretability is key, especially for ensemble models like XGBoost and LightGBM. Furthermore, these methods can be plugged into the real-world development environment, IDEs, or CI/CD pipelines, to get real feedback. Moreover, advanced data imbalance solutions beyond SMOTE and SMOTEENN can enhance the detection rate for rare code smells, while automatic feature selection and dimensionality reduction can improve the performance of the detection rate on a hand while reducing the computational load. With dynamic analysis and static analysis, we will be able to have a better idea of how many code quality issues are in our dataset, and the more diverse a dataset gets, the more robust and accurate our models will be because they are based on as much different code as possible. Taken together, these research directions seek to increase the scope of code smell detection, increase interpretability, ensure real-world applicability, improve data management techniques, and increase model efficiency so that automated code quality assessment tools can be improved.

5.6 Conclusion

On the final note, this research has shown how ensemble learning and advanced resampling methods can combine very effectively with Python code smell detection, to handle the "Large Class" and "Long Method". We can accurately identify code smells that can have a big harm on the software maintainability and quality with promising results using robust ML models and some great ideas to handle data imbalance. The results indicate that ensemble methods, such as SMOTE, can improve performance in accepting that code is of a higher quality.

This study also shows the impact of model interpretability, real-world applicability, and the need for diverse datasets for a more robust code smell detection system. With the evolution of software development practices, these methodologies must be continuously refined exploring cross-language applications, other types of code smells, and integrating static and dynamic analysis. Future research should go further to devise better tooling that not only points to code smells but also gives developers some actionable insights, towards better software quality and more effective upkeep. This research provides the foundation to extend automated code quality assessment tools to become more reliable and generalizable in real-world software development contexts.

References

[1] Sandouka, Rana, and Hamoud Aljamaan. "Python code smells detection using conventional machine learning models." PeerJ Computer Science 9 (2023): e1370.

[2] Dewangan, Seema, et al. "A novel approach for code smell detection: an empirical study." IEEE Access 9 (2021): 162869-162883.

[3] Arcelli Fontana, Francesca, et al. "Comparing and experimenting machine learning techniques for code smell detection." Empirical Software Engineering 21 (2016): 1143-1191.

[4] Kovačević, Aleksandar, et al. "Automatic detection of Long Method and God Class code smells through neural source code embeddings." Expert Systems with Applications 204 (2022): 117607.

[5] Fontana, Francesca Arcelli, and Marco Zanoni. "Code smell severity classification using machine learning techniques." Knowledge-Based Systems 128 (2017): 43-58.

[6] Pushpalatha, M. N., and M. Mrunalini. "Predicting the severity of open source bug reports using unsupervised and supervised techniques." Research Anthology on Usage and Development of Open Source Software. IGI Global, 2021. 676-692.

[7] Vatanapakorn, Natthida, Chitsutha Soomlek, and Pusadee Seresangtakul. "Python code smell detection using machine learning." 2022 26th International Computer Science and Engineering Conference (ICSEC). IEEE, 2022.

[8] Dewangan, Seema, and Rajwant Singh Rao. "Method-Level Code Smells Detection Using Machine Learning Models." International Conference on Computational Intelligence in Pattern Recognition. Singapore: Springer Nature Singapore, 2022.

[9] Dewangan, Seema, et al. "Code smell detection using ensemble machine learning algorithms." Applied sciences 12.20 (2022): 10321.

[10] Yadav, Pravin Singh, et al. "Machine learning-based methods for code smell detection: a survey." Applied Sciences 14.14 (2024): 6149.

[11] Farasat, Talaya, and Joachim Posegga. "Machine Learning Techniques for Python Source Code Vulnerability Detection." arXiv preprint arXiv:2404.09537 (2024).

[12] Ratnawati, Fajar, and Jaroji Jaroji. "Decision Tree for Smell Code Detection in Python: A Practical Implementation." Proceedings of the 11th International Applied Business and Engineering Conference, ABEC 2023, September 21st, 2023, Bengkalis, Riau, Indonesia. 2024.

[13] Chen, Zhifei, et al. "Detecting code smells in Python programs." 2016 international conference on Software Analysis, Testing and Evolution (SATE). IEEE, 2016.

[14] Yao, Xiaotong, Xiaoli Fu, and Chaofei Zong. "Short-term load forecasting method based on feature preference strategy and LightGBM-XGboost." IEEE Access 10 (2022): 75257-75268.

[15] Everton Gomede. “Synthetic Minority Over-Sampling Technique (SMOTE): Empowering AI through Imbalanced Data Handling.” Medium, 11 Mar. 2024, pub.aimind.so/synthetic-minority-over-sampling-technique-smote-empowering-ai-through-imbalanced-data-handling-d86f4de32ea3. Access Date: 04 November 2024

[16] Yang, Fangyuan, et al. "A hybrid sampling algorithm combining synthetic minority over-sampling technique and edited nearest neighbor for missed abortion diagnosis." BMC Medical Informatics and Decision Making 22.1 (2022): 344.

[17] Ren, Qiuyi. "Research on Encrypted Text Classification Based on Natural Language Processing." Journal of Physics: Conference Series. Vol. 1792. No. 1. IOP Publishing, 2021.

[18] Tynykulova, Assemgul, et al. "Integrating numerical methods and machine learning to optimize agricultural land use." International Journal of Electrical & Computer Engineering (2088-8708) 14.5 (2024).

[19] Liu, Yaning, et al. "Prediction of protein crotonylation sites through LightGBM classifier based on SMOTE and elastic net." Analytical biochemistry 609 (2020): 113903.

[20] Valentina. “Understanding AdaBoost for Decision Tree.” Medium, 31 Jan. 2020, towardsdatascience.com/understanding-adaboost-for-decision-tree-ff8f07d2851. Access Date: 03 November 2024

[21] javaTpoint. “Machine Learning Decision Tree Classification Algorithm - Javatpoint.” Www.javatpoint.com, 2021, www.javatpoint.com/machine-learning-decision-tree-classification-algorithm. Access Date: 04 November 2024

[22] Devinterview-io. “GitHub - Devinterview-Io/Random-Forest-Interview-Questions” GitHub, 2024, github.com/Devinterview-io/random-forest-interview-questions?tab=readme-ov-file. Accessed 4 Nov. 2024.

[23] Tantithamthavorn, Chakkrit, et al. "An empirical comparison of model validation techniques for defect prediction models." IEEE Transactions on Software Engineering 43.1 (2016): 1-18.

[24] Chicco, Davide, and Giuseppe Jurman. "The advantages of the Matthews correlation coefficient (MCC) over F1 score and accuracy in binary classification evaluation." BMC genomics 21 (2020): 1-13.

[25] Chicco, Davide, Matthijs J. Warrens, and Giuseppe Jurman. "The Matthews correlation coefficient (MCC) is more informative than Cohen's Kappa and Brier score in binary classification assessment." Ieee Access 9 (2021): 78368-78381.

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

[27] Chawla, Nitesh V., et al. "SMOTE: synthetic minority over-sampling technique." Journal of artificial intelligence research 16 (2002): 321-357.

[28] Yang, Fangyuan, et al. "A hybrid sampling algorithm combining synthetic minority over-sampling technique and edited nearest neighbor for missed abortion diagnosis." BMC Medical Informatics and Decision Making 22.1 (2022): 344.

[29] Chen, Tianqi, and Carlos Guestrin. "Xgboost: A scalable tree boosting system." Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining. 2016.

[30] Ke, Guolin, et al. "Lightgbm: A highly efficient gradient boosting decision tree." Advances in neural information processing systems 30 (2017).

[31] Schapire, Robert E. "Explaining adaboost." Empirical inference: festschrift in honor of vladimir N. Vapnik. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013. 37-52.

Attachment

- Source Code: https://github.com/jfemon8/Thesis

- Datasets: https://github.com/jfemon8/Thesis/tree/main/Dataset