IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 11 October 2023, accepted 22 October 2023, date of publication 25 October 2023, date of current version 1 November 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3327553

RESEARCH ARTICLE

A Severity Assessment of Python Code Smells

AAKANSHI GUPTA[1], RASHMI GANDHI[1], NISHTHA JATANA[2], DIVYA JATAIN[2], SANDEEP KUMAR PANDA[3], AND JANJHYAM VENKATA NAGA RAMESH[4]

[1]Department of Computer Science and Engineering, ASET, AUUP, Noida 201303, India
[2]Maharaja Surajmal Institute of Technology, Delhi 110058, India
[3]Department of Artificial Intelligence and Data Science, Faculty of Science and Technology (IcfaiTech), The ICFAI Foundation for Higher Education, Hyderabad, Telangana 501203, India
[4]Koneru Lakshmaiah Education Foundation, Vijayawada, Andhra Pradesh 522502, India

Corresponding author: Sandeep Kumar Panda (sandeeppanda@ifheindia.org)

This work was supported in part by the Faculty of Science and Technology (IcfaiTech), The ICFAI Foundation for Higher Education, Hyderabad, Telangana, India.

ABSTRACT Presence of code smells complicate the source code and can obstruct the development and functionality of the software project. As they represent improper behavior that might have an adverse effect on software maintenance, code smells are behavioral in nature. Python is widely used for various software engineering activities and tends tool to contain code smells that affect its quality. This study investigates five code smells diffused in 20 Python software comprising 10550 classes and analyses its severity index using metric distribution at the class level. Subsequently, a behavioral analysis has been conducted over the considered modification period (phases) for the code smell undergoing class change proneness. Furthermore, it helps to investigate the accurate multinomial classifier for mining the severity index. It witnesses the change in severity at the class level over the modification period by mapping its characteristics over various statistical functions and hypotheses. Our findings reveal that the Cognitive Complexity of code smell is the most severe one. The remaining four smells are centered around the moderate range, having an average severity index value. The results suggest that the J48 algorithm was the accurate multinomial classifier for classifying the severity of code smells with 92.98% accuracy in combination with the AdaBoost method. The findings of our empirical evaluation can be beneficial for the software developers to prioritize the code smells in the pre-refactoring phase and can help manage the code smells in forthcoming releases, subsequently saving ample time and resources spent in the development and maintenance of software projects.

INDEX TERMS Software maintenance, code smell severity, cognitive complexity code smell, class change proneness, open-source software, Python, sustainable software.

I. INTRODUCTION

The code quality of the software is a significant factor that majorly contributes to software maintenance. Beck et al. [1], [2] introduced the concept of code smells, which are blips in software code due to its improper software design and development by the application programmer. These mainly emerge from developer actions taken at times of emergencies, careless implementation or by using subpar coding solutions. It is believed that code smells, being design flaws, harm the quality of the code [1], [2]. Conceptually, they classify the violations in software that follow object-oriented design approaches such as data abstraction, encapsulation, modularity, and hierarchy [3]. Moreover, code smells are also considered Technical Debt (TD). The refactoring process boosts the performance of the software code [4]. Since refactoring some smells can be an expensive and time-consuming affair [5], therefore, the priority of smells needs to be realized in the early stages only. The prioritization of code smell can also be represented as a severity index, as proposed by Fontana et al. [6].

Apart from the severity of code smells, during the software releases, the changes in the software modules or classes also need to be dealt with. Experts frequently look at multiple updates to get historical information regarding software project development. Early recognition of modules that inculcate smells can be valuable for the team handling software maintenance and assigning substantial testing resources with a higher probability of alteration in software modules. Software-changing impact analysis has been previously

The associate editor coordinating the review of this manuscript and approving it for publication was Porfirio Tramontana.

performed by some researchers by association rule mining [7]. Change-prone module prediction is an active research area attracting many researchers and is a crucial software maintenance activity helping to prioritize code smells for refactoring [8].

Hence, the motivation of this study drills deep into the design issues of the Python software, particularly the code smells, for better maintainability and reusability of the code. This research has been studied to investigate the severity of the code smells diffused in the Python language. Alongside this, a behavioral analysis has been performed by evaluating the class change proneness over the severity of the smells estimated. This behavior analysis is evaluated over three phases collectively considered in this study under the term modification period: Phase 1: initial development version, Phase 2: mid-modification version, and Phase 3: latest released version.

Moreover, statistical analysis has also been performed for the above-analyzed behavior of the code smells over the considered phases to obtain better insights into the Python code at the class level. Consequently, this research work benefits in evaluating the diffusion of code smell over the modification period and evaluating it statically, which a developer can exercise for an efficient development environment and progressive maintenance life cycle.

Research Contributions of this work are as follows:

- Assessing the diffusion of Python code smells.
- Behavioral study of Python code smell severity based on the class change proneness over the metric distribution of the relevant code smell.
- Performance comparison of various multinomial classifiers on the estimated code smells of the severity of the latest released Python software using the AdaBoost (Adaptive Boost) Boosting method.
- Investigating the class change proneness for code smell severity in different phases of modification period using statistical analysis -Kruskal Wallis Hypothesis Test and Wilcoxon Signed Rank Test.

The rest of the paper is organized as follows:

Section II presents a detailed background study and motivation of the work; Section III presents the Experimental Setup with code smell severity analysis and research questions discussion, and Section IV briefs about the threats to validity. Finally, Section V reviews the conclusions of the work and presents the prospects of working in the area.

II. BACKGROUND

This section provides detailed background and related work. The existence of code smells is associated with code quality issues such as code modifiability & understandability, which can further lead to maintenance issues in the software. Code smells are signs of poor code design choices or the use of shortcuts while coding, which can lead to a higher defect rate and lower software quality [4], [6]. Accurately evaluating maintainability based on code smells requires understanding their criticality and ability to reflect software aspects important for maintainability and their limitations. While detection & removal of smells have been extensively researched in the past decade, the impact on software performance and maintainability still needs to be fully understood. This forms the motivation for the work presented in this paper.

Smells do not always harm the code immediately, but they are a threat and can cause software development issues. Tofani et al. [9] stated that the code smell is one of the symptoms of poor design in technical debt and practical implementation alternatives. Over the years, many projects concluded that the evaluation code smells are introduced. Moreover, specifying the requirement may be code smells degrading the software quality and efficient outcome for both evaluation and maintenance. There are many existing techniques of code anomalies ranking in the evolution era. Development does not remove the code smells but emphasizes severe code smells. Hence, to lessen the duplication, smoothly running the application will leverage past programming efforts.

Code smells can be rectified by using appropriate refactoring techniques [1], [2]. The refactoring techniques work by exposing the design flaws that are affecting the code maintainability. Code smells of static languages such as C++ and Java have already been explored widely. Beck et al. [1], [2] proposed 22 generic code smells that are language-independent. Some researchers manifest the manual detection of code smells, while others have proposed different detection tools such as infusion, JDeodorant, iPlasma, PMD and JSNose [10]. JSNose is the earliest detection tool regarding dynamic languages for code smells. It can identify both additional and generic in JavaScript code [11]. The rapid usage of Python language for the development of large projects across various domains has brought new performance requirements to be dealt with. The testability, maintainability and understandability issues need to be carefully monitored. Vatanapakorn et al. [12] proposed a machine learning-based model for code smell detection. They trained their model on 115 open-source Python software, 22-functions- and 39-class-level project metrics. Their model could achieve 99.27% accuracy and could outperform the tuning machine method.

Holkner and Harland [13] investigated the dynamic nature of Python programs to study whether dynamic activities are prompted when a Python program starts executing. Chen et al. followed a constraint-based approach to identify type-related bugs in Python programs. A Python smell detection tool, Pysmell, was introduced by Chen et al. [14] to produce smell reports for Python programs.

Some approaches have been proposed by researchers who studied machine learning algorithms, such as classification or associative mining for detecting smells. Khomh et al. [15] specify smells on the basis of Bayesian belief networks that consider the detection process's inherent unpredictability. Polamba et al. [16] developed a textual-based method to identify bad code smells by calculating the likelihood that a specific smell will impact a component. Additionally, they proposed a method for detecting smells using

TABLE 1. Some relevant studies with the considered python code smells.

S.NO.	AUTHOR	TITLE	SUMMARY	NOVELTY AND MAJOR CONTRIBUTION
1.	Z. Chen et al. [14]	Understanding metric-based detectable smells in Python software: A comparative study.	Researchers defined ten code smells in this research and developed a metric-based recognition approach with three distinct filtering strategies, viz., Statistics-Based Strategy, Experience-Based Strategy, and Tuning Machine Strategy. These strategies specify metric thresholds & the findings highlight the distinctive qualities of Python smells.	To specify metric thresholds, this work built a metric-based detection approach with three distinct filtering strategies, viz., Statistics-Based Strategy, Experience-Based Strategy, and Tuning Machine Strategy and introduced ten code smells. The effectiveness of three detection algorithms in detecting Python smells is compared, as well as how these smells affect program maintainability using various detection strategies.
2.	Dewangan et al. [19]	A novel approach for code smell detection: an empirical study	In this research, six machine learning algorithms are applied to four different code-smell datasets viz., God-class dataset, Feature- envy, and Long-method dataset and Data-class dataset, to predict the code smells. These datasets are generated from 74 open-source systems. To detect the best metrics for improving accuracy, Chi-square and Wrapper-based feature selection technique is used. Further, to improve the accuracy, Grid search algorithm is applied for parameter optimization and improving accuracy of all the applied algorithms.	The novelty of this work lies in its two-step approach, wherein in the first step, different machine learning algorithms are used for detecting the code smells. To detect the best metrics for improving accuracy, Chi-square and Wrapper-based feature selection technique is used. Further, to improve the accuracy, Grid search algorithm is applied for parameter optimization. Results show that for Data class dataset, while considering all features, the highest accuracy value is achieved by Random Forest algorithm (99.74%). The worst performance while considering all features is shown by Naive Bayes algorithm (83.10%)
3.	Cao et al. [20]	Towards better dependency management: A first look at dependency smells in python projects.	The main aim of the researchers in this work is to avoid the introduction of code smells that frequently happens while managing cross project dependencies. Using ill written and ill maintained dependency configuration files is major reason for the occurrence of this issue. The researchers specifically work on three dependency smells in Python	The researchers contributed by developing & implementing a tool (Python Cross-project Dependency- PyCD) that uses configuration files for extracting dependency information.

TABLE 1. *(Continued.)* **Some relevant studies with the considered python code smells.**

			Projects, viz., Bloated Dependency, Version Constraint Inconsistency and Missing Dependency.	This tool outperforms the other state-of-art methods for the 212 projects selected by the authors. An empirical study is carried out to identify the prevalence, evolution and reasons for the existence of the three dependency smells in 132 Python projects.
4.	Chen et al. [21]	Evaluating test quality of Python libraries for IoT applications at the network edge.	The researchers identified that the IoT applications are quite vulnerable at the network edge where the frequent exchange of data is frequently carried out. This fact, combined with the difficulty of statically analysing the dynamic Python libraries which are generally used in IoT development, makes the issue even worse & affects the performance. So, the researchers proposed a framework for identifying test quality for the IoT- Python libraries.	The researchers proposed a framework called Pysta that uses hybrid analysis engine for multi-dimensional evaluation of Python test code. Conducting an empirical study on some of the most frequently used Python libraries for IoT development & test code evaluation, the authors identify the major problems and challenges such as low-test coverage in large-scale projects, low naming consistency, delaying test co-evolution, lacking edge tests, single form of assertion and various code smells in tests. This research intends to improve the quality of IoT applications while reducing the associated security risks.
5.	Vatanapakorm et al. [12]	Python Code Smell Detection Using Machine Learning	The researchers used a dataset of 115 open-source Python projects, 22 function-level software metrics and 39 class-level software metrics to train eight machine-learning models. According to the findings, the machine learning algorithm identified long methods and long base class lists with an accuracy of 99.72%.	This study suggests a machine learning-based method for Python programs to identify code smells. With a dataset comprising 115 open-source Python projects, 22 function-level software metrics and 39 class-level software metrics, the authors trained eight machine-learning models to recognise different code smells at both the class and function levels. This method used logistic
				regression-forward stepwise (conditional) selection and Correlation-based feature selection (CFS) to enhance the According to the

TABLE 1. *(Continued.)* **Some relevant studies with the considered python code smells.**

				findings, the machine learning algorithm identified long methods and long base class lists with 99.72% accuracy. The machine learning-based way of detecting code smells also beats the tuning machine approach. It was also discovered that a certain collection of high-impact traits can distinguish each sort of code smell.
6.	Gupta A. et al. [22]	Prioritising Python Code Smell for Refactoring Using MCDM	This research used the Multi-Criteria Decision-Making (MCDM) approach to identify a ranking order of considered code smells. This resulted in the realisation of the priority for an efficient & cost-effective reworking procedure.	Refactoring the code to redesign it while maintaining its functionality is the best way to address code smell. This research used the Multi-Criteria Decision-Making (MCDM) approach to identify a ranking order of considered code smells. This resulted in the realisation of the priority for an efficient & cost-effective reworking procedure.

co-changes obtained from the version to detect five smells [17]. Kessentini et al. [18] introduced a cooperative parallel search-based approach where different approaches to code smell detection were integrated. The hybridization of distinct strategies in a parallel optimized way generates detection rules for structural metrics code smell. Table 1 represents some recent works most relevant in Python code smells.

There are significant secondary studies published in the area that aim to comprehensively analyze the existing literature on code smells [23], [24].

A. CODE SMELL SEVERITY

Software design quality metrics only administer a hint about software design quality. However, metrics need to be sufficiently semantically rich to enable users (e.g., developers and code maintainers) to understand what is going wrong and what can be a possible solution. On the other hand, code smells identify more complex and semantically rich structures in source code and help detect design issues using software metrics. Moreover, the degree to which a smell in the code can impact software maintainability is pensive. Uncovering many code quality issues due to smells in the code, parallelly influences the criticality of smells in the software world from the refactoring phase. Realizing the severity of smell, among others, gives the developer community an edge for optimization at the earliest.

A tool designed for ranking code smells was suggested in 2015 by Vidal et al. [25]. The tool uses a blend of three factors: historical changes to components (for stability assessment), critical system modifiability scenarios, and the significance of the code smell. Its value is subjective as the developer can indicate how harmful the smell is. The threshold for identifying a code smell may differ across developers and systems. The researchers propose a tool called SPIRIT, that ranks the severity of code smells in a system, considering its criticality. They evaluated the approach in two case studies and found the results helpful to developers. Another method Fontana proposed involves applying strong and weak filters to reduce the number of code smell detection outcomes. However, this method is limited to code smells of only five types [26].

Ratiu et al. used historical data of suspected flaw-related structures as a metric-based detection strategy to express code smells regarding thresholds [27]. Other studies have also explored the use of system history to predict classes that are likely to change in the future based on those that have frequently changed in the past [28]. In a different approach, Zhao et al. [29] proposed a hierarchical method for identifying and prioritizing refactoring opportunities based on their

predicted improvement to software maintainability. While analyzing the two systems, they did not establish an Intensity Index for code smells.

Fontana et al. [6] have previously presented a study similar to the presented research, where they introduced the Code Smelly Intensity Index as a criterion for prioritizing code smells. Our study also incorporates this approach, although Fontana's method was limited to a specific set of code smells. An approach has been followed in this study that addresses exploring the Python code smells and analyzing their behavior (change proneness) over a modification period, thereby focusing on the criticality of the smell through severity intensity values, which can help the developers realize the importance of particular smell in pre refactoring phase thus, saving time and resources.

Code smells are researched in several languages, including Java, C, C++, Kotlin, etc. To the best of our knowledge, plenty of content doesn't address Python code smells. Given how frequently Python is used these days, the behavior of code smells in Python software needs to be addressed. This study aims to evaluate the severity of Python code smells so they can be promptly fixed or removed.

III. METHODOLOGY/ WORKFLOW OF THE WORK

This research escalates the study of the diffusion of code smells in Python software for severity assessments at the class level. A glimpse of the workflow of the research has been depicted in Figure 1. The deployment of the proposed strategy is carried out on a suitable and valid dataset of 20 Python software (open-source software) with 10,550 classes that are taken from the GitHub repository. The most frequently occurring code smells were then selected. After the software has been extracted, an examination of static code metrics is carried out to obtain statistical information about the software. Further the code smell severity evaluation process has been performed to prioritize the code smells on behalf of their severity index.

In detail the 20 Python software has been studied and extracted using the GitHub repository in the following manner:

- Severity estimations and analyzing the best algorithm for multinomial classification of the obtained severity ranges in Python software, considering the latest released Python software (Phase 3 software)
- Behavioral analysis of the severity of diffused code smells and statistical exploration in Python software over the phases (Phase 1, Phase 2, Phase 3).

Formally, 20 Python software with 10,550 classes were analyzed for detecting Python code smell according to their occurrences at the class level. These classes were then employed to estimate the severity of five considered code smells rooted in the metric distribution of the analysed software systems.

While speculating on the factors concerning the Python code smells, the essence of the study is summarized in three research questions. These research questions are mentioned below, along with their brief motivation:

RQ1: Which classification techniques best estimate the severity of the code smells affecting Python software?

In RQ1, the classification of the severity of Python code smells is performed by the application of various algorithms, studied under supervised machine learning techniques, mainly multinomial classification [12], [31]. The classification is supported by the severity computation method, which yields the severity ranges in terms of the severity index [32], [34].

RQ2: How does the severity of the Python code smell behave over the modification period?

With this research question, an attempt has been made to examine the severity of code smells over the modification period by estimating the class change proneness over the metric distribution of the relevant code smell [14], [31]. Further, it will help to determine the percentage change of the severity with respect to the initial software development, considered the first version of the modification period in this study [22].

RQ3: Can the smells be prioritized based on the diffusion of their severity in software code metrics?

This research question explores the mean rankings of the severity of the code smell estimated in the above approach [19], [20]. Subsequently, the versions are monitored to ascertain the class-by-class behaviour of the severity of different smells [14].

A. DATASET AND SMELL SELECTION AND COLLECTION

This research has been carried out using open-source Python software. Among the substantial Python software available, 20 codes were evaluated, with approximately 6,817 Python files and 10,550 Python classes.

The source codes implemented by these systems are freely accessible and can be downloaded and copied from the GitHub worldwide repository of open-source software. The primary criterion for the selection of the systems is their reputation & acceptance on the GitHub platform, which can be demonstrated by MOST STARS or as STARGAZERS (The Stargazers software are the software that is highly popular among the users, widely used, and is most reliable, making them a suitable choice for the research study). The relevant material related to this research is uploaded on Github[1].

This research studies code smells detected by SonarQube, which violate the code's design structure and hinder the code's maintainability and readability. Code smell assessment and choice of criteria are based on specific factors. The nature and impact of a specific code smell comes first, followed by the frequency with which practitioners encounter it while working.

Listed below are the smells that are considered in this work.

- Cognitive Complexity
- Collapsible "if" statements
- Many Parameters List

[1] https://github.com/Aaashi21/Python.git

- Naming Conventions
- Unused Variable

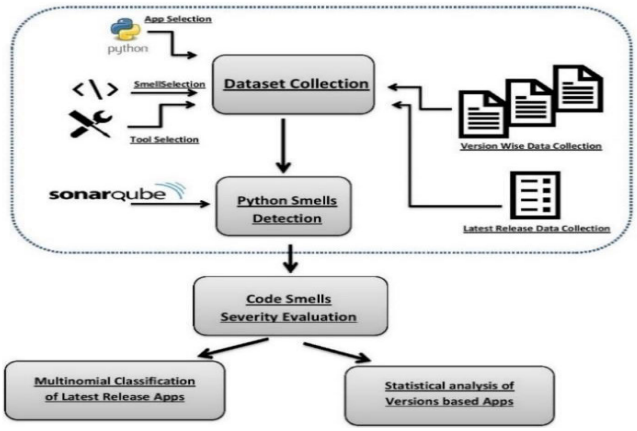

FIGURE 1. Workflow of the study.

These smells can be detailed as follows:

1. Cognitive Complexity: Cognitive Complexity is a metric designed specially to measure understanding and comprehensibility. It can be described as the number of independent dimensions concept worthy; the individual brings to bear in mind describing a particular domain. It is accused based on an object-sorting task. Its adequacy has been measured on five criteria: Independence from verbal abilities, High-level test-retest reliability, Association with other indices of developed social cognition, and association with measures of developed social cognition. Overall, it is clear that it is a measure of understanding regarded as unsatisfactory [34].
2. Collapsible "If" statements: This refers to a coding practice where multiple if statements that are nested together can be combined into a single if statement with compound conditions. This is being done to improve code readability and reduce redundancy. When nested if statements perform similar checks or share common conditions, collapsing them into a single statement can make the code more concise and easier to understand [35].
   Example: if condition1:
   If condition2:
   do_something()
   #collapsed version
   If condition1 and condition2
   do_something()
3. Many Parameters List: Many parameters lists in a code function, often referred to as "Long parameter lists" can lead to code smells. Their code smells make the code harder to understand, maintain and test. They can result in poor readability, increased complexity and reduced flexibility. It is advisable to keep the number of parameters in a function to a minimum, ideally no more than 3-4. As they recognise this code smell, they consider techniques like refactoring to simply the functions interface, using a data structure to group related parameters, or applying the builder pattern to improve the code quality [35].
4. Naming Convention: Naming conventions in software are important for writing readable, maintainable, and understandable code. Otherwise, it raises potential code quality issues, which are often related to naming. Common issues include meaningless/confusing frames, inconsistent naming, overly long names, and generic names. Common naming conventions used in software development are Camel case, Pascal case, Smoke case, and Kebab case. Using one (or more) of these conventions may contribute to better collaboration, reduced confusion, and smoother code integration. Hence, it is important to standardize naming conventions for proper efficacy [36].
5. Unused Variable: Unused variables are a common code smell programming, indicating that variables have been declared; it is not being used or referenced anywhere else in the code. This can lead to confusion, misunderstanding, and inability to maintain code. Common sources of unused variables are incomplete [1].

The Python classes for each software that are diffused with code smells indicate the presence of respective smells and are termed TRUE, whereas the rest of the classes (absence of smell) are marked as FALSE. SonarQube tool is one of the most popular industrial tools for source code technical debt measurement and is preferred by various studies for code quality inspection purposes. Figure 2 represents the diffusion of code smells at the class level for the 10,550 Python classes considered in this work.

B. CODE SMELL SEVERITY ANALYSIS

The process of analyzing the severity of code smells has been carried out by evaluating a numeric severity value using the metric distribution obtained from analyzing the software through a static code analyzer, Understand tool. The Python software was quantitatively analyzed to obtain software metrics using UNDERSTAND software versioned BUILD-978 (https://scitools.com/). While inspecting the code, 37 attributes were obtained. The explained process below for the Code smell severity analysis would yield a code severity index value, like the one obtained in the study by Fontana et al. [6]. Moreover, this computation is conducted for class-level instances of the analyzed Python software, which have diffused code smells.

For intensity computation, the following steps are followed:

Step 1: First, analyze the software code quantitatively in the form of custom metrics and procure the dominant attributes. Depending upon the metric distribution of each attribute, thresholds are evaluated. Further, different thresholds derived for software design metrics are computed. The computed threshold values acknowledge the metric's statistical properties.

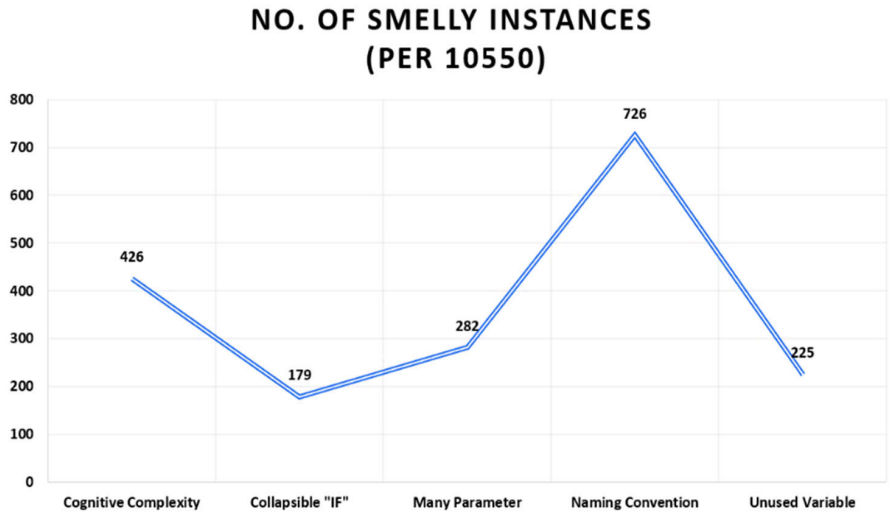

FIGURE 2. Diffusion of code smells at the class level.

The severity, as stated earlier, is examined for the smell-causing instances. Based upon the starting threshold point of the smelly instances and the extreme threshold point, four intervals are observed and treated over the metric distribution. These thresholds are called relative threshold points related to the metric values corresponding to smelly instances.

The relative threshold percentile, mapped to their respective values, also rests on the Comparator (>, <), referred to as the Absolute Comparator, for determining the side of the metric distribution where the probability of occurrence of smell is high. These comparators were obtained by applying a rule-based classification algorithm, particularly JRIP, on the essential software metrics extracted using feature selection methods.

Step 2: The metric values are then compared with their threshold values obtained for the defined percentile. The intensity levels are defined as follows:

[1-3.25): INFORMATIONAL
[3.25-5.5): LOW
[5.5-7.75): MODERATE
[7.75-10): MAJOR
[11]: CRITICAL

*INFORMATIONAL and CRITICAL are considered extreme thresholds.

Step 3: The last step heads towards aggregating the intensity values obtained for each metric to obtain a single value at the class level by employing the Absolute Comparator obtained in step 1(e).

These steps would retrieve relative threshold values of every metric for the considered percentiles. This threshold-driven approach assists in assigning ranges to the obtained values, which in turn describes the criticality of the metric values.

IV. RESULTS

The extent to which the code smell can severely affect the Python software has been further segmented into a set of research questions. These research questions are explained in detail with all the processes applied and the results obtained as follows:

RQ1: Which of the classification techniques performs best for estimating the severity of the code smells affecting Python software?

RQ1 inspects the best multinomial classifier, accurately classifying the severity of code smells diffused in Python software. This severity has been obtained as a *code smell severity index* using the method described *in the Code smell Severity Analysis* section. The dataset prepared for analysis comprises a set of 10,550 classes, and the latest versions of Python software are considered (Phase 3).

TABLE 2. Code smell severity index.

Python Code Smells	Severity
Cognitive Complexity	8.01
Collapsible 'IF'	7.37
Many Parameter	7.02
Naming Conventions	6.22
Unused Variable	7.07

A. CLASSIFICATION BASED MODEL SELECTION

For classifying the severity intensities obtained above, the severity index obtained for each smell has been categorized into a range and taken as a labelled category for further multinomial classification modelling. Using the mentioned intensity levels in the Code Smell Severity Analysis section, it was observed that out of five code smells, one of the smells happened to lie in the Major range with an intensity value of 8.01. The remaining four are centered around the Moderate range, averaging 7.

Furthermore, the acquired dataset has been tested against various multinomial supervised machine learning algorithms

TABLE 3. Software metrics considered along with absolute comparator.

Python Based Code Smells	Absolute Comparator '>'	Absolute Comparator '<'
Cognitive Complexity	AvgLineCode CountClassCoupled MaxCyclomaticModified	CountLineCode MaxNesting
Collapsible 'IF'	SumEssential CountLineCodeExe CountLineComment MaxCyclomaticStrict CountDecInstanceVariable	MaxNesting CountStmtExe
Naming Conventions	CountStmt SumCyclomaticModified AvgLineCode CountLineCodeDecl RatioCommentToCode	CountDecMethodAll MaxInheritanceTree
Many Parameter	AvgLine CountLineComment MaxCyclomaticModified SumEssential CountDeclMethodAll CountStmt CountDeclInstanceVariable	CountDecInstanceMethod CountLineBlank
Unused Variable	CoutnStmt CountDecInstanceVariable CountStmtExe	AvgLineBlank MaxInheritanceTree

TABLE 4. Performance of multinomial classifier based on accuracy.

Python Code Smells	J48	GRIP	KNN	Random Tree	Naïve Bayes
Cognitive Complexity	98.5	97.8	86.6	96.7	68.7
Collapsible 'IF'	92.1	89.9	80.4	85.4	34
Many Parameter	84.3	86.1	76.1	78.6	49.4
Naming Convention	97.2	97.6	92.5	95.3	74
Unused Variable	92.8	92	74.6	87.1	54.6
Average	**92.98**	**92.68**	**82.04**	**88.62**	**56.14**

TABLE 5. Performance measure of J48 algorithm.

Python Code Smells	Accuracy	Precision	ROC Area	F-Measure	Kappa Statistics
Cognitive Complexity	98.59%	98.6	99.1	98.6	0.9586
Collapsible 'IF'	92.17%	92.3	98.7	92.2	0.8576
Many Parameter	84.34%	84.3	92.8	84.2	0.7046
Naming Conventions	97.24%	97.3	99.7	97.2	0.961
Unused Variable	92.88%	93.1	97.3	92.9	0.8747

for each smell individually. Multinomial classification is performed by consideration of the nominal variable. The available implementations of classifiers allow both multinomial and binary classification. These classifiers are combined with the 'ADA Boost' ensemble technique.

The Ada boost is an adaptive boosting strategy applied in machine learning as an ensemble method. It initializes the weights by reallocating each instance with higher weights to handle uncategorized data. Boosting is a supervised approach that reduces biases and variance. It behaves iteratively in combination with weak classifiers to achieve stable solutions that provide a robust output. Boosting is a general ensemble method that is used to create a robust classifier from a few weak classifiers. The multinomial categories for which the

severity data has been classified have been labelled from the intensity scale division per the severity computation analysis.

For further severity analysis, feature selection techniques were used to find the essential software metrics and lower the dimensionality of data using different methods. The dataset of individual code smell was evaluated for the following combinations of feature selection approaches: *Feature Selector: Information Gain*; *Searching Approach: Ranker*. Table 2 depicts the Code smell severity index estimated at the class level. The software metrics listed in Table 3 were obtained using the above feature selection combination, and their comparator was evaluated using the JRIP algorithm. Additionally, the metrics are chosen on behalf of code complexity, cohesion, coupling, and size [30]. In this regard, Tree-based classifiers like J48, Random Tree, JRIP, Naïve Bayes, and K-Nearest Neighbor (KNN) are the classification algorithms considered over which the severity computation for each code smell has been efficiently trained, tested, and validated. J48 is a decision tree algorithm used in machine learning and data mining. Trees are formed by recursively partitioning the data into subsets based on the values of different attributes, aiming to create a tree structure used for classification tasks and known for its simplicity and effectiveness. It can also deal with the characteristics, missing attributes data estimations and varying attribute costs. All the similar code smells are placed at one position if the instances belong to similar cases. It computes the info gained at each node in the data and will be considered from the tested attribute list. Finally, the best attribute is selected by computing the selection parameters. There are some limitations, as the computed info gain value may make the tree wider and more complicated, generating different subsets, which may lead to overfitting. The performance evaluation has been executed using the 10-fold cross-validation.

After evaluating different classifiers, J48 (Decision Tree classifier), combined with ADABOOST, is the best-performing classifier, with an average accuracy of 92.98%, as described in Table 4. Other performance measures of the obtained classifier, in combination with ADABOOST learning, with their respective values, are shown in Table 5. To select subsets of code smell rather than focusing on complete sets, feature selection is being performed. To select the code smells, J48 and Adaboost are used as they are versatile and can be integrated for outlier detection, regression, and classifier. This integration handles the title data and is being trained to support weak classifier results. Although Adaboost may not perform well with noisy data and is slow to train, hybridization with J48 makes it easier to make the subset, while classifiers are not able to decide fast as the tree size is wider individually. As the literature states, J48 faces difficulty handling empty subsets, avoidable code smells, and overfitting the data being Adaboost.

The RQ1 acknowledges the severity of Python code smells and the best-suited multinomial classification algorithm for inspecting the criticality of code smells using the ensemble technique (ADA Boost) through supervised learning approaches.

RQ2: How does the severity of the smell behave over the modification period?

The analysis of the Python software concerning code smells and their severity compelled the researchers of this work to inspect the behavior of code smells for the prior versions of the software by analyzing its behavior over the period. The following criterion was devised for the selection of different versions of software in the form of ''Phases'' under the name of modification period considered in this study:

Phase 1: The software's initial version is considered the development version with no modifications.

Phase 2: The mid version of the software during its maintenance phase is considered an in-modification version.

Phase 3: The latest software release version is considered the recently modified version.

Once the data has been prepared for all the phases, the changes have been evaluated for Phase 2 and Phase 3 with respect to Phase 1. Moreover, for analyzing the behavior of severity of code smells, Phase 1 is kept constant for further estimations as it reflects the initial development cycle that might further be improved in later phases through immediate refactoring. This assumption helps to observe the changes over the modification period.

These three phases were further analyzed for 20 Python software at the class level, and common classes were evaluated between the versions. Out of the approximately 20K classes, only 899 were found to be in sync with all three versions. Hence, this dataset of 899 classes was then analyzed for predicting the class change proneness based on the severity of considered code smell over the modification period.

Figure 3 discusses the distribution of considered code smells among the considered versions over the common classes. From the observations, the evolution of Python software considered across the common classes observed over the modification period approximates a 62.7% change from Phase 1 to Phase 2 and a 69.3% change from Phase 1 to Phase 3.

Before estimating the class change proneness based on the severity of code smells, it was required to identify the classes affected by the considered smells. The detection of code smells has been implemented using the SonarQube platform, based on a static code metric analyzer (Understand Tool), likewise, RQ1, but this time, the detection was laid for all three phases independently. The detection of code smells inferred that around 8-9% of classes were diffused by ''Cognitive Complexity'' smell, 2-4% by ''Collapsible IF'' smell, 3-5% by ''Long Parameter List'' smell, 4-9% by ''Naming Convention'' smell and 2-3% by ''Unused Variable'' smell when observed over the modification period (Phase 1, Phase 2, Phase 3). After detection, it was necessary to extract the vital software metrics and then labelling was done for the presence and absence of respective code smells.

Additionally, a feature selection approach was applied in this investigation, like that in RQ1.

Feature Selector: Information Gain, Gain Ratio, CFS Subset Evaluator;

Searching Approach: Ranker, Best First Greedy.

The rule-based classification technique was applied to the acquired dataset of Version 1 to obtain the vital software metrics along with their Absolute Comparator.

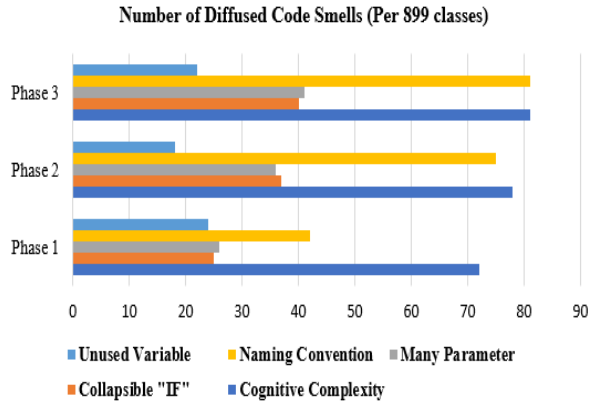

FIGURE 3. Distribution of considered code smells among the considered versions.

The process of estimating the severity index was ultimately practiced, similar to RQ1, on all three phases of the considered code, smells. Thus, the evaluated severity index has been described in Table 6 for each considered version over the modification period. It can be articulated that the severity index of Collapsible 'IF' smell, Naming Convention smell and Unused Variable smell have progressive severity observed over the modification period. On the contrary, Cognitive Complexity smell and many parameters list smell show a decline in severity over the phases.

TABLE 6. Code smell Severity index over the modification period.

Python Code Smells	Phase 1	Phase 2	Phase 3
Cognitive Complexity	7.93	7.75	7.73
Collapsible 'IF'	7.07	7.1	7.27
Naming Convention	6.03	6.19	6.15
Many Parameters List	6.58	6.38	6.02
Unused Variable	6.43	6.43	6.71

The highest rise was computed for Collapsible 'IF' code smell, accounting for 2.8%. In contrast, many Parameters code smell indicates a drop of 8.5%. The smells Collapsible ''IF'' and Naming Convention show an incremental severity over the modification period w.r.t severity of Phase 1. The other smells, Many Parameters List and Cognitive Complexity, show a decremental severity. However, the Unused Variables code smell, with a percentage change of 4.19%, is not considered due to its consistent severity behavior observed over the modification period, Phase 1– Phase 2.

Consequently, this analysis promotes an ideology among the developers to reduce the severity of the Python code in the later versions of the software exhibiting code smells. Likewise, it would benefit the developers laying a foundation in Python development.

The RQ2 analyses the behavior of Python code smells concerning the severity of the code smells by estimating the class change proneness across the versions analyzed over the modification period.

RQ3: Can the smells be prioritized based on the diffusion of their severity in Python software?

Yes, the smells can be prioritized based on the diffusion of their severity in the Python software. While computing the class change proneness based on the severity of code smells, the result revealed that their severity intensity values are approximately similar when evaluated over the modification period. However, when it was evaluated for different code smells, we still got approximately similar results, though all code smells are contrasting. With this view, an attempt has been made to rank the severity intensity of different code smells for the class-by-class estimation of severity intensity changes. In this study, two hypothesis tests were applied to the data for statistically analyzing the behavior of severities of code smells for all three phases:

- Kruskal Wallis H Test
- Wilcoxon Signed Rank-Test

Kruskal Wallis H test- The Kruskal-Wallis H test is a rank-based nonparametric test that is used to determine a significant statistical difference for two or more groups of an independent variable (McKnight 2010). In this part of the research, the independent group belongs to the different code smells under observation, possessing severity in close proximity. The code smells were prioritized using the severity intensity discussed above, and each code smell was labelled using a numeric value (0-4) corresponding to the intensity ranges. The dataset used here refers to the severity index obtained for Phase 1 Python software.

Hypothesis:

H*0*: The distribution of samples of the severity of each code smell originates from an identical population.

H*1*: The distribution of samples of the severity of at least one code smell comes from a different population than the others.

***Procedure*:**

1. The severity index of Version 1 for each code smell is initially ranked from N groups. Version 1 is considered due to its initial development nature. Also, it has been utilized for calculating the changes in the further versions.
2. The test statistics followed on the data distribution of intensity values of code smells are given by:

$$T = \frac{(N-1)(S_t^2 - C)}{S_r^2 - C} \tag{1}$$

where,

ni = number of observations in group i

S_i^2 = Average Rank of all observations in group i:

$$S_i^2 = \sum_{i=1}^{k} \frac{R_i^2}{n_i} \tag{2}$$

S_r^2 = Rank of observations j from group i:

$$S_r^2 = \sum_{i=1}^{N} r_{ij}^2 \tag{3}$$

C= Average of all the rij:

$$C = \frac{N(N+1)^2}{4} \tag{4}$$

3. The decision to reject or accept the null hypothesis is made by comparing it to a critical value obtained for a given significance or alpha level.

In this investigation, the alpha level obtained is 0.00, which is less than 0.05. Thus, the null hypothesis is rejected, concluding that the distribution of all the severity values obtained for the considered code smell is not identical. The severity distribution of considered code smells evaluated at the class level has been visualized as box plots in Figure 4., exhibiting some outliers. The code smells are denoted by a numeric value ranging from 0 to 4 for *Cognitive Complexity, Collapsible 'IF', Many Parameters, Naming Convention,*and *Unused Variable*, respectively.

A pair-by-pair comparison has been drafted from observing the mean ranks of Kruskal Wallis hypothesis testing. Figure 5. illustrates the comparison between the mean rankings of the considered code smells having a significance value of less than 0.05. The blue line adjoining pair represents a strong significance pairwise rejection of the hypothesis, whereas the red line represents a weak significance of rejection of the hypothesis.

Wilcoxon Signed Rank Test- The severity index obtained for Phase 1 software has been tested and prioritized using the Kruskal Wallis test among the different code smells, as explained above. Further, it was required to compare the results across the modification period (Phase 1, Phase 2, Phase 3). For this, Wilcoxon signed rank test was preferred. This test has been performed two times as there are two different pairs to be compared. The pairs compared through this test are Phase 1- Phase 2 and Phase 1- Phase 3.

Hypothesis:

H_0: The distribution of performance of the two versions is equal, i.e., the median of their difference is 0.

The distribution of performances of the pair of versions is unequal, i.e., the media of their difference $\neq$ is 0.

Test Statistics:

Let N be the number of pairs. Thus, there are a total of 2N data points for the considered versions of each code smell.

For pairs, i =1…N, let x_1, i and x_2, i denote the measurements.

W = The sum of signed ranks.

$$W = \sum_{i=1}^{Nr} [sgn\left(x_{2,i} - x_{1,i}\right) .R_i \tag{5}$$

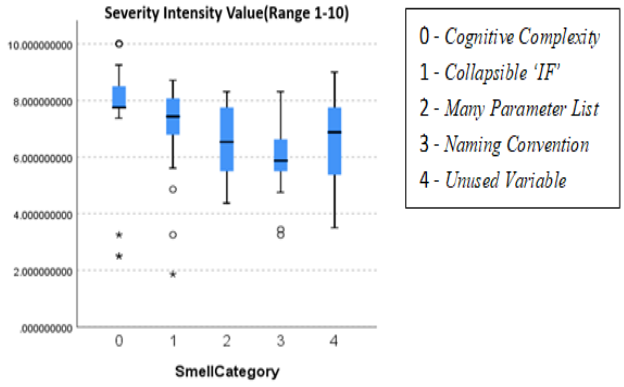

FIGURE 4. Box plot of the distribution of severity of considered code smells.

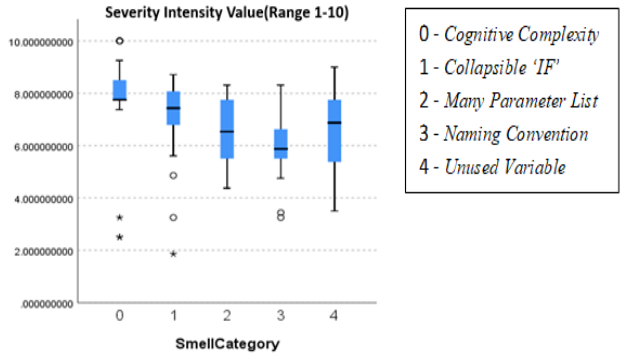

FIGURE 5. Pairwise comparison of code smells based on mean rankings obtained through the Kruskal Wallis test.

where,

sgn= sign function

Nr= reduced sample size

Ri = Rank of the pair, starting with the smallest non-zero absolute difference.

[Note: Ties receive a rank equal to the average of the ranks they span.]

For Nr $\geq$ 20, The z- score can be computed as:

$$z = \frac{W}{\sigma_w} \tag{6}$$

where,

$$\sigma_w = \frac{\sqrt{Nr(Nr+1)(2Nr+1)}}{6} \tag{7}$$

By performing the Wilcoxon test for the severity intensity values at class-level, the code smells exhibiting an incremental nature (*Collapsible "IF"*and *Naming Convention*), as described in RQ2, possess more positive classes than the negative ones in terms of severity for the pair *Phase 1 - Phase 2* and *Phase 1 - Phase 3.* Moreover, through this analysis, these smells prevail in an incremental nature of severity intensities, as concluded in RQ2. On the contrary, the code smells having a decremental nature was observed to have more negative classes over the modification period. This signifies an optimized performance of the software,

which has minimized the effect of code smells and renders the users with optimal functionalities in various software systems developed in Python.

These estimations have been described in Figure 6. for the "*Naming Convention*" smell and Figure 7. for the "*Cognitive Complexity*" smell. The mean rankings for the positive and the negative classes have been illustrated in Table 7 for "*Many Parameter*" (Phase 1-Phase 2) and Table 8 for "*Collapsible IF*"" (Phase 1- Phase 3).

This infers that most of the classes of these smells should be refactored at the earliest in the subsequent versions of the Python software to avoid maintainability and software quality issues. This concludes that some of the classes diffused with these smells have been refactored in the subsequent software versions, yielding good software quality.

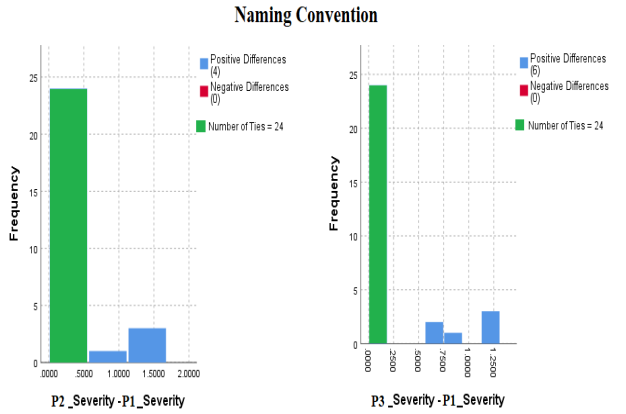

FIGURE 6. Class-level study of naming convention smell.

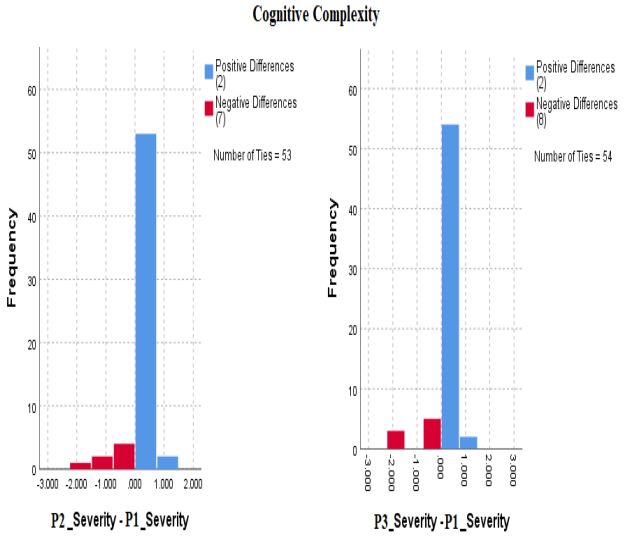

FIGURE 7. Class-level study of cognitive complexity smell.

TABLE 7. Mean ranking of many parameters among Phase 1 – Phase 2.

	Many Parameter			
Phases		N	Mean Rank	Sum of Ranks
	Negative Ranks	7[a]	6.21	43.50
P2_Severity - P1_Severity	Positive Ranks	5[b]	6.90	34.50
	Ties	9[c]		
	Total	21		
a. P2_Sev < P1_Sev	b. P2_Sev > P1_Sev	c. P2_Sev = P1_Sev		

TABLE 8. Mean ranking of collapsible 'If' among Phase 1 – Phase 3.

Collapsible 'IF'				
Phases		N	Mean Rank	Sum of Ranks
	Negative Ranks	3[a]	6.67	20.00
P3_Severity - V1_Severity	Positive Ranks	11[b]	7.73	85.00
	Ties	4[c]		
	Total	18		
a. P3_Sev < P1_Sev	b. P3_Sev > P1_Sev	c. P3_Sev = P1_Sev		

V. THREATS TO VALIDITY

In this section, the potential threats associated with the progress of this study have been examined. The primary concern of this study revolves around the identified code smells. Only five specific smells from the Python software have been considered in analyzing their severity across the latest release and over the modification period. The inclusion of other code smells could potentially affect the results. Additionally, the effectiveness of the dataset utilized could potentially impede the outcomes.

It should be noted that the severity assessment is based solely on the distribution of metrics for the considered Python software system and the establishment of thresholds. As severity estimation is subjective, other factors may be considered to prioritise smells. The feature selection methodologies utilised may be further refined by assessing the weights of different metrics. Using a single detection tool may impact the identification of code smells across the classes of software systems, and the evaluation has been conducted solely at the class level without considering method-level data. Also, some code smell scenarios are very challenging to detect, thus there is a potential that some code smells will go undetected.

VI. CONCLUSION & FUTURE WORK

This research implements multinomial machine-learning classification algorithms for Python code smells. Out of

five code smells, the Cognitive Complexity smell falls under the major range with a severity index of 8.01. The remaining four smells are centered around the moderate range, averaging 7. The estimated class level severity intensity for each code smell was then tested for performance comparison of multinomial classifiers in combination with AdaBoost and outcomes with 92.8% accuracy for the J48 algorithm.

The analysed three phases estimate the class change proneness of the considered 20 Python software systems and produce a total of 899 common classes among them. The changes between the software metrics range from 2% to 9% for the considered code smells among the versions. It has been established that a comparison of the software versions with respect to Phase 1 represents an incremental nature of severity for three code smells, namely ''Collapsible IF'' and ''Naming Convention''. In contrast, the other smells, ''Cognitive Complexity'' and ''Many Parameters,'' have been observed to have decreasing values of severity index over the period. However, the ''Unused Variable'' smell shows an abrupt nature.

Hypothesis testing using the Kruskal Wallis Test and Wilcoxon Signed Rank test has been implemented for analyzing the differences in the distribution of the smells. The obtained mean ranks through Kruskal Wallis hypothesis testing reveal that each code smell differs from the other, expressing a non-identical population distribution among the code smells.

The study of Wilcoxon signed Rank Test states that ''Collapsible 'IF' and Naming Convention code smell possess more positive classes. However, the smells ''Cognitive Complexity'' and ''Many Parameter'' have been observed to have more negative classes, implicating a reduced severity value observed during estimations over the modification period. This concludes that some of the classes diffused with these smells have been refactored in the subsequent versions of the software, yielding good software quality. Despite the changes observed in severity at the class level for the ''Unused Variable'' smell, the examination revealed less significant changes over the modification period.

The novelty of this work is attributed to the severity analysis of the Python code smells using J48 and Adaboost algorithm. The work includes severity analysis of Python code smells, which has not been addressed in the available literature. The contribution of this work would help the software developers prioritize the code smells in the pre-refactoring phase, thus saving ample time and resources spent in the development of projects. Subsequently, examining the behavior of the severity trend of code smell gives a glimpse to the developers for managing the code smells in the forthcoming releases for the primary software system. This work encourages the researchers to further explore Python code smells to explore the diffusion and criticalness among the smells. In addition, co-occurrences of Python smells can be explored by prioritising them and exploring the critical smells to be refactored simultaneously.

REFERENCES

[1] K. Beck, M. Fowler, and G. Beck, ''Bad smells in code,'' in *Refactoring: Improving the Design of Existing Code*, vol. 1, 1999, pp. 75–88.

[2] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Reading, MA, USA: Addison-Wesley, 2018.

[3] E. V. d. P. Sobrinho, A. De Lucia, and M. d. A. Maia, ''A systematic literature review on bad smells-5 W's: Which, when, what, who, where,'' *IEEE Trans. Softw. Eng.*, vol. 47, no. 1, pp. 17–66, Jan. 2021.

[4] F. A. Fontana and M. Zanoni, ''Code smell severity classification using machine learning techniques,'' *Knowl.-Based Syst.*, vol. 128, pp. 43–58, Jul. 2017.

[5] T. Sharma and D. Spinellis, ''A survey on software smells,'' *J. Syst. Softw.*, vol. 138, pp. 158–173, Apr. 2018.

[6] F. A. Fontana, V. Ferme, M. Zanoni, and R. Roveda, ''Towards a prioritization of code debt: A code smell intensity index,'' in *Proc. IEEE 7th Int. Workshop Manag. Tech. Debt (MTD)*, Oct. 2015, pp. 16–24.

[7] L. Moonen, T. Rolfsnes, D. Binkley, and S. Di Alesio, ''What are the effects of history length and age on mining software change impact?'' *Empirical Softw. Eng.*, vol. 23, no. 4, pp. 2362–2397, Aug. 2018.

[8] V. Lenarduzzi, N. Saarimäki, and D. Taibi, ''Some SonarQube issues have a significant but small effect on faults and changes. A large-scale empirical study,'' *J. Syst. Softw.*, vol. 170, Dec. 2020, Art. no. 110750.

[9] A. Tofani, A. Di Pietro, L. Lavalle, M. Pollino, and V. Rosato, ''CIPRNet decision support system: Modelling electrical distribution grid internal dependencies,'' *J. Polish Saf. Rel. Assoc.*, vol. 6, no. 3, pp. 133–140, 2015.

[10] X. Liu and C. Zhang, ''DT: A detection tool to automatically detect code smell in software project,'' in *Proc. 4th Int. Conf. Machinery, Mater. Inf. Technol. Appl.*, Jan. 2016, pp. 681–684.

[11] A. Gong, Y. Zhong, W. Zou, Y. Shi, and C. Fang, ''Incorporating Android code smells into Java static code metrics for security risk prediction of Android applications,'' in *Proc. IEEE 20th Int. Conf. Softw. Quality, Rel. Secur. (QRS)*, Dec. 2020, pp. 30–40.

[12] N. Vatanapakorn, C. Soomlek, and P. Seresangtakul, ''Python code smell detection using machine learning,'' in *Proc. 26th Int. Comput. Sci. Eng. Conf. (ICSEC)*, Dec. 2022, pp. 128–133.

[13] A. Holkner and J. Harland, ''Evaluating the dynamic behaviour of Python applications,'' in *Proc. 32nd Australas. Conf. Comput. Sci.*, vol. 91, 2009, pp. 19–28.

[14] Z. Chen, L. Chen, W. Ma, X. Zhou, Y. Zhou, and B. Xu, ''Understanding metric-based detectable smells in Python software: A comparative study,'' *Inf. Softw. Technol.*, vol. 94, pp. 14–29, Feb. 2018.

[15] F. Khomh, M. Di Penta, and Y.-G. Gueheneuc, ''An exploratory study of the impact of code smells on software change-proneness,'' in *Proc. 16th Work. Conf. Reverse Eng.*, Oct. 2009, pp. 75–84.

[16] F. Palomba, G. Bavota, M. Di Penta, F. Fasano, R. Oliveto, and A. De Lucia, ''A large-scale empirical study on the lifecycle of code smell co-occurrences,'' *Inf. Softw. Technol.*, vol. 99, pp. 1–10, Jul. 2018.

[17] F. Palomba, G. Bavota, M. D. Penta, R. Oliveto, D. Poshyvanyk, and A. De Lucia, ''Mining version histories for detecting code smells,'' *IEEE Trans. Softw. Eng.*, vol. 41, no. 5, pp. 462–489, May 2015.

[18] W. Kessentini, M. Kessentini, H. Sahraoui, S. Bechikh, and A. Ouni, ''A cooperative parallel search-based software engineering approach for code-smells detection,'' *IEEE Trans. Softw. Eng.*, vol. 40, no. 9, pp. 841–861, Sep. 2014.

[19] S. Dewangan, R. S. Rao, A. Mishra, and M. Gupta, ''A novel approach for code smell detection: An empirical study,'' *IEEE Access*, vol. 9, pp. 162869–162883, 2021.

[20] Y. Cao, L. Chen, W. Ma, Y. Li, Y. Zhou, and L. Wang, ''Towards better dependency management: A first look at dependency smells in Python projects,'' *IEEE Trans. Softw. Eng.*, vol. 49, no. 4, pp. 1741–1765, Apr. 2023.

[21] Z. Chen, C. Jia, and L. Chen, ''Evaluating test quality of Python libraries for IoT applications at the network edge,'' *Wireless Netw.*, 2023, doi: 10.1007/s11276-023-03479-2.

[22] A. Gupta, D. Sharma, and K. Phulli, ''Prioritizing Python code smells for efficient refactoring using multi-criteria decision-making approach,'' in *Proc. Int. Conf. Innov. Comput. Commun.*, 2022, pp. 105–122.

[23] J. Prabhu, T. Guhan, M. A. Rahul, P. Gupta, and M. S. Kumar, ''An analysis on detection and visualization of code smells,'' in *Artificial Intelligence for Sustainable Applications*. Hoboken, NJ, USA: Wiley, 2023, pp. 163–176.

[24] M. Jerzyk and L. Madeyski, "Code Smells: A comprehensive online catalog and taxonomy," in *Developments in Information and Knowledge Management Systems for Business Applications*, vol. 7. Cham, Switzerland: Springer, 2023, pp. 543–576.
[25] S. A. Vidal, C. Marcos, and J. A. Díaz-Pace, "An approach to prioritize code smells for refactoring," *Automated Softw. Eng.*, vol. 23, pp. 501–532, Dec. 2014.
[26] F. A. Fontana, V. Ferme, and M. Zanoni, "Poster: Filtering code smells detection results," in *Proc. IEEE/ACM 37th IEEE Int. Conf. Softw. Eng.*, vol. 2, May 2015, pp. 803–804.
[27] R. Marinescu and D. Ratiu, "Quantifying the quality of object-oriented design: The factor-strategy model," in *Proc. 11th Work. Conf. Reverse Eng.*, Nov. 2004, pp. 192–201.
[28] D. Silva, N. Tsantalis, and M. T. Valente, "Why we refactor? Confessions of GitHub contributors," in *Proc. 24th ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, Nov. 2016, pp. 858–870.
[29] Z. Li, T.-H. Chen, J. Yang, and W. Shang, "DLFinder: Characterizing and detecting duplicate logging code smells," in *Proc. IEEE/ACM 41st Int. Conf. Softw. Eng. (ICSE)*, May 2019, pp. 152–163.
[30] *Understand by SciTools*. [Online]. Available: https://documentation.scitools.com/pdf/metricsdoc.pdf
[31] A. Martins, C. Melo, J. Monteiro, and J. Machado, "Empirical study about class change proneness prediction using software metrics and code smells," in *Proc. 22nd Int. Conf. Enterprise Inf. Syst.*, 2020, pp. 140–147, doi: 10.5220/0009410601400147.
[32] A. Gupta and N. K. Chauhan, "A severity-based classification assessment of code smells in Kotlin and Java application," *Arabian J. Sci. Eng.*, vol. 47, no. 2, pp. 1831–1848, Feb. 2022, doi: 10.1007/s13369-021-06077-6.
[33] F. A. Fontana, V. Ferme, M. Zanoni, and A. Yamashita, "Automatic metric thresholds derivation for code smell detection," in *Proc. IEEE/ACM 6th Int. Workshop Emerg. Trends Softw. Metrics*, May 2015, pp. 44–53, doi: 10.1109/WETSoM.2015.14.
[34] W. A. Scott, "Cognitive complexity and cognitive flexibility," *Sociometry*, vol. 25, no. 4, pp. 405–414, Dec. 1962, doi: 10.2307/2785779.
[35] R. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice, NJ, USA: Prentice-Hall, Jan. 2008.
[36] A. Hunt and D. Thomas, *The Pragmatic Programmer*. Reading, MA, USA: Addison-Wesley, 2000.

AAKANSHI GUPTA received the B.Tech. and M.Tech. degrees in computer science and engineering discipline, and the Ph.D. degree from USICT, GGSIPU, Delhi. She is currently with the Amity School of Engineering and Technology, Noida, and affiliated with AUUP, Noida. She has more than 13 years of teaching and research experience. Her research interests include software engineering, software designing, data mining, and machine learning algorithms.

RASHMI GANDHI received the Ph.D. degree from GGSIPU. She has been an Assistant Professor with CSE, ASET, AUUP, Noida, since 2008. She is certified as an oracle certified professional (OCP) in NVIDIA courses. She has many research publications (more than 25) in renowned Scopus conferences and journals. She is a passionate and creative teacher committed to being a catalyst in the discovery of one's potential, calling and passion, which ultimately impact, equip, and empower the present and future generations.

NISHTHA JATANA received the B.Tech. degree from the Computer Science and Engineering Department, the M.Tech. degree in computer technology and applications, and the Ph.D. degree in software testing from Guru Gobind Singh Indraprastha University, in 2021. She is currently an Associate Professor with the Computer Science and Engineering Department, Maharaja Surajmal Institute of Technology. She has more than 12 years of teaching and research experience. She has various research publications, including SCIE-indexed journal articles, book, and conference papers published in various ESCI/Scopus publications. Her research interests include software testing, meta-heuristic approaches and network security, and assistive technologies.

DIVYA JATAIN is currently pursuing the Ph.D. degree with the Department of Computer Science and Applications, CDLU. She is also with the Maharaja Surajmal Institute of Technology, New Delhi. She has a teaching experience of more than 11 years. Her research interests include data analysis and data analytics, social networks analysis, natural language processing, big data, and emerging technologies.

SANDEEP KUMAR PANDA is currently working as a Professor and the Head of the Department of Artificial Intelligence and Data Science, Faculty of Science and Technology (IcfaiTech), ICFAI Foundation for Higher Education (Deemed to be University), Hyderabad, Telangana, India. He has published 50 papers in international journals and international conferences and book chapters in repute. He has 17 Indian patents on his credit. He has six Edited books named *Bitcoin and Blockchain: History and Current Applications* (CRC Press, USA), *Blockchain Technology: Applications and Challenges* (Springer ISRL), *AI and ML in Business Management: Concepts, Challenges, and Case Studies* (CRC Press), *The New Advanced Society: Artificial Intelligence and Industrial Internet of Things Paradigm* (Wiley Press, USA), *Recent Advances in Blockchain Technology: Real-World Applications* (Springer ISRL), and *Metaverse and Immersive Technologies: An Introduction to Industrial Business and Social Applications* (Wiley Press), in his credit. He has ten lakhs seed money projects from IFHE. His research interests include blockchain technology, the Internet of Things, AI, and cloud computing. He received the Research and Innovation of the Year Award 2020 from MSME, Government of India and DST, Government of India, New Delhi, in 2020, and the Research Excellence Award from Brand Honchos, in 2022. He is a Reviewer of IEEE Access. His professional affiliations are MIEEE, MACM, and LMIAENG. He also received the "Best Teacher Award", and Cash Prize of Rs. 1 Lakh from The ICFAI Foundation for Higher Education, Hyderabad, Telangana, India, on 13th Convocation 2023.

JANJHYAM VENKATA NAGA RAMESH is currently an Assistant Professor with the Department of CSE, Koneru Lakshmaiah Education Foundation, Vaddeswaram, Andhra Pradesh, India. He is having 20 years of experience in teaching for UG and PG engineering students. He has published more than 25 papers in IEEE/SCI/Scopus/WoS journals and conferences. He has authored six text books and ten book chapters. His research interests include wireless sensor networks, computer networks, deep learning, machine learning, and artificial intelligence. He is a reviewer of various leading journals.