Arabian Journal for Science and Engineering
https://doi.org/10.1007/s13369-021-06077-6

RESEARCH ARTICLE-COMPUTER ENGINEERING AND COMPUTER SCIENCE

A Severity-Based Classification Assessment of Code Smells in Kotlin and Java Application

Aakanshi Gupta[1] · Nidhi Kumari Chauhan[1]

Received: 31 October 2020 / Accepted: 12 August 2021
© King Fahd University of Petroleum & Minerals 2021

Abstract
Code smells instigate due to the consistent adoption of bad programming and implementation styles during the evolution of the software which adversely affects the software quality. They are essentially focused and prioritized for their effective removal based on their severity. The study proposed a hybrid approach for inspecting the severity based on the code smell intensity in Kotlin language and comparing the code smells which are found equivalent in Java language. The research work is examined on five common code smells that are complex method, large class long method, long parameter list, string literal duplication, and too many methods over 30 open-source systems (15 Kotlin/15 Java). The experiment compares different machine learning algorithms for the computation of human-readable code smell detection rules for Kotlin, where the JRip algorithm proved to be the best machine learning algorithm with 96% and 97% of overall precision and accuracy, validated at 10-fold cross-validation. Further, the severity of code smell at the class level is evaluated for prioritization of applications written in Kotlin and Java language. Moreover, the process of severity computation is semiautomated using the CART model, and thus, metric-based severity classification rules are achieved. The experimentation provides a complete understanding of prioritization of code smells in Kotlin and Java and helps to attain prioritized refactoring which will enhance the utilization of resources and minimize the overhead rework cost.

Keywords Code smells · Severity · Machine learning · Prioritization · JRip · Classification and regression trees (CART)

1 Introduction and Motivation

Modern application development has been drastically commutated over the years. In recent times, the software development market has been seen emerging and furnished with several advancements in programming languages with the aim of reducing the efforts and resources of the development team [1]. The developers' adoption of such advancements does not guarantee success in delivering maintainable software [2]. Subsequently, it is well understood that most of the software development time is consumed by the maintenance phase of the software development life cycle (SDLC) [3]. However, the development team attempts to develop an optimal software product, but many times, the software maintainability is hampered for various reasons [4]. The main reason for such hampering on which several practitioners focused their studies are code smells or sometimes referred to as bad smells or just smells [5].

Kent Beck and Martin Fowler first used the word code smell, and according to Martin Fowler, code smell is a surface indication that usually corresponds to a deeper problem [6]. Code smell may be regarded as the source code's inefficiencies, which may lower overall software performance and increase software maintainability problems [7]. They mainly emanate from the development team's adoption of low-coding practices and poor implementation choices in the software evolution process [3, 8]. Accordingly, they are not bugs and do not prevent the software from its actual action but pinpoints carelessness and shortcut methods adopted by the developers, which may affect the yielding competence in the future [8]. They should be premediated from the very early phase in the software evolution process to sustain maintainability issues concerning technical debt [9].

✉ Aakanshi Gupta
aakankshi@gmail.com

Nidhi Kumari Chauhan
nidhikchauhan0411@gmail.com

[1] Department of Computer Science and Engineering, Amity School of Engineering and Technology, Guru Gobind Singh Indraprastha University, New Delhi, India

Published online: 01 September 2021

Technical debt was defined by Ward Cunningham, which accounts for all the additional rework costs needed to keep software maintainable and up to date [10]. Conceptually, refactoring was seen as an integral method for removing the code smells [6, 11]. Before endorsing any such method type, it is equally imperative to determine the detrimental effect of each code smell, and it is evident that different types of code smell had different impacts on the codebase [12]. Thus, prioritizing each code smell in advance based on their harmfulness will surely help developers to enhance their conventional refactoring course to a prioritized refactoring action. The application of such action will undoubtedly assist in choosing which code smell demand high urgency removal and will quickly best refurbish high maintainable software in a given limited period [13, 14].

Formerly, the Java programming language has been extensively used for development, until in 2017, Google announced its first-class support for Kotlin, which was developed by JetBrains [15–18]. Kotlin is a powerful open-source cross-platform, statistically typed, and general-purpose programming language that combines both object-oriented and functional programming features [19, 20]. Kotlin seems slightly different from Java, but it is compatible with all the existing Java codes, frameworks, and libraries [3, 21]. Kotlin provides an easy way to switch from Java to Kotlin by installing a plugin to the respective IDE (Integrated Development Environment) [16, 21]. Further, Kotlin includes smart extensions and provides modern development features as compared to Java [15, 16, 22]. Though occupied with several advantages, Kotlin lacks experienced and widespread developer communities resulting in limited learning resources and more time in finding solutions to problems [15, 23]. Despite some shortcomings, Kotlin was seen still emerging with more and more developers tends to choose Kotlin as a source of powerful, equipped development language. In 2018, Kotlin was seen as the fastest growing language on GitHub, with approximately 2.6 times more developer adoption rate than the previous year [20].

Nevertheless, previous studies investigated the quality of android applications written in the Kotlin language [16]. Bose et al. [15] provide a contrast between Java and Kotlin based on advancements and new features. Mathews et al. [3] proposed a relative study between Java and Kotlin and found that Kotlin programs have fewer code smells than Java programs. Emil et al. [19] perform a study for the perception and effects of implementing Kotlin in existing projects. However, to the best of our knowledge, they did not approach the topic in terms of severe code smells that needed to be refactored first and whose impact hamper the software recklessly. The major portion of work done is focused on different languages and their respective domains in which Java holds a high position [24]. Therefore, Java has been extremely explored already in the field of code smells, whereas a little amount of work is done in Kotlin regarding the same [15]. Hence, Kotlin, as an evolving language, requires a tremendous amount of attention and research exploration [20].

The paper combines two individual approaches to ultimately ignite and undertake the removal of the harmful nature of critical code smells. This research work targets five different code smells, which are common in both Kotlin and Java. The primary focus of the research is on the extraction of rules for the detection of code smells in Kotlin language and further finding the severity of common code smells in Kotlin as well as in Java. In view of this goal, a term named severity index is defined, which measures the severity of each code smell based on static code metrics. The estimation of the severity index is computed using the hybrid approach. The severity index is used for the evaluation of the harmfulness of the code smells within different smells in the same language (inter-smell intra-language) and between the same code smells in a different language (intra-smell inter-language). Ultimately, prediction of varying severity level is achieved in the form of metric-based severity classification rules, which are supposed to work for rooted detection of severity level for estimating the scattering of respective code smell in the codebase.

Hence, the motivation of the paper is to drill down the unexplored, and unheard facts about the domain of code smell in Kotlin and Java language and help to replace the commonly used tactics with new enhancements so that resources are not exploited anymore. Thus, the research is centered on finding answers to four different research questions that laid the foundation for this study.

The research questions, along with their motivation, are as follows:

RQ1 Is there any method for developers to detect the code smells in Kotlin, and how accurately do the code smells follow it?

This preliminary research question attempts to discover a method for the developers' advancements to identify the code smells in Kotlin systems, which can operate as a replacement to the conventional detection strategies. The proposed method uses machine learning algorithms for detection, which are evidenced by different performance measures to produce accurate results.

RQ2 Can a methodology be formulated for prioritization of code smell at coarse granularity level (class level) for Kotlin applications?

The improvement in the quality of the software is simultaneously related to the quality of refactoring, which is done to remove the code smell from the source code. The quality of refactoring is enhanced when we ascertain our focus from less severe to more severe code smell. This aspect demands a customized approach for studying and comparing the severe effect of code smell at the lowest possible level and providing which code smell and class require immediate action course. Thus, the research question proposes a methodology termed as the hybrid approach which is used for the prioritization of code smells in Kotlin applications.

RQ3 Can the common code smells be prioritized in Java applications for achieving effective refracting using hybrid approach?

The research question studies for evaluating the severity of common code smells found in Java language. Further, showing which code smell possess high severity and should be prioritized for refactoring based on the common code smells severity index.

RQ4 Is it possible to semiautomate the hybrid approach for an in-depth classification to predict the severity level based on software code metrics?

The methodology applied for the computation of the severity index of code smell requires constant human support and resources. The need for optimization of the hybrid approach used suggested a semiautomation for the prediction of severity level, which is achieved with the help of the classification process and will surely assist the developers in evaluating the severity level by a much more refined and automated process.

Therefore, the significant contributions of the research are as follows:

- Investigated the common code smells in Kotlin and Java language.
- Applied various machine learning algorithms and generated the most efficient human-readable code smell detection rules in Kotlin language.
- Applied the hybrid approach for evaluation of severity index for each considered code smell of Java and Kotlin.
- Prioritization of the code smell for most frequent removal from the source code.
- Applied classification and regression trees (CART) for in-depth evaluation of different severity levels by obtaining *metric-based severity classification rules*.

The paper is structured in the following sections: Sect. 2 illustrates the background study, whereas Sect. 3 represents the proposed methodology carried out in the research work, which explains the experimentation process in detail. Section 4 presents the result analysis using various statistical graphs. Section 5 describes the threats to the validity of the work done. Finally, Sects. 6 and 7 provide the conclusions and future scope of the work, respectively.

2 Background Study

The software developers mainly consider the functionality which needs to be embedded into the evolving software that needs to be delivered to the demanding clients in a limited amount of time. It is studied that most of the development processes are governed by the time factor rather than focusing on optimal software delivery. These small-scale differences do not temporarily affect the software, but once they start impairing the software, it may head to future failures and technical debt. This section is organized as follows: Sect. 2.1 illustrates the evolution and detection strategies of code smells. Section 2.2 describes the eradication of code smells from the systems. Section 2.3 depicts the approaches followed for the prioritization of code smells. Section 2.4 illustrates the comparison studies between Kotlin and Java.

2.1 Code Smell Evolution and Detection

Almost every software today is prone to code smells one way or the other, if it is not taken care of from the developing stage [8]. The term code smell was first accomplished by Kent Beck and further encouraged Martin Fowler [6]. Sharma et al. [22] analyzed that the code smells mainly evolve due to the lack of development skills, and when more emphasis is laid on advanced features rather than on software quality. Tufano et al. [4] studied the reasons for bad smells that occurred in the software and their survivability over the change history of 200 open-source projects from different software ecosystems. Kumar et al. [25] explained the challenges and explored the best practices which can highly optimize the mobile application development process. Hetch et al. [26] tracked the quality of software along their development process. Several studies conform to different detection mechanisms for unmasking the code smells. Gupta et al. [1] experiment that a supervised machine learning algorithm can be used for the identification of detection rules for android code smells. Interestingly, very less work in Kotlin complements automated detection of code smells using static code metrics. Ideally, only two static analysis tools are available for Kotlin and which are integrable with different IDEs [3]. Fontana et al. [27] introduced a data-driven method to derive threshold value for code metrics, which can be used for obtaining detection rules for code smells. The most popular techniques for code smell detection comprises software metrics that are accomplished by focusing on different threshold extraction frameworks [28]. Guggulothu et al. [29] come up with a multi-label approach for code smell detection and whether the given code element is affected by many smells. Hozano et al. [30] use developer feedbacks for improving the detection of code smells. Palomba et al. [24] provide a tool for automated detection of android code smells. Pecorelli et al. [31] considered five code smell types and compare the machine learning model with Décor, a state-of-the-art heuristic-based approach.

2.2 Code Smell Removal

Developers dealing with code smell detection also try to approach their removal techniques. The eviction of code smells from the codebase helps us to properly optimize the use of the detection process and reduce the future rework cost on the software [2, 32]. Many research works argue that there are still chances for improvements in the detection and removal of code smells detection [33]. The main emphasis is laid on refactoring, which is observed as a responsible parameter for discharging code smells from the software. Today, investigators and evaluators mainly concentrate on the automated removal of code smells rather than on manual removal techniques because automated techniques require fewer resources and produce highly optimized results. Fowler et al. [6] released a handbook mentioning different techniques of refactoring for improving the design of existing code. Saika et al. [34] presented that software developers only concentrate on particular types of smells, and refactoring does not decrease the severity of code smells. Kessentini et al. [35] also seemed for detection and refactoring using an automated tool called refactory. Cruz et al. [36] proposed an automated tool-based refactoring approach for improving the energy consumption of android applications using Leafactor. Palomba et al. [11] performed manual refactoring for analyzing the energy leaks in android applications caused by nine code smells.

2.3 Code Smell Prioritization Approaches

The refactoring of the most severe code smells assists the software to increase the quality and performance initially by dealing with the most harmful code smells in the codebase. Zhang et al. [37] investigated the relationship between six code smells and software faults and prioritized the refactoring process using code bad smells. Vidal et al. [13] presented a semiautomated approach to concentrate on the most severe issues in the structural design of the software using a tool that suggests the priority of code smells based on past component modifications, critical modifiability scenarios for the system, and relevance of the kind of smell. Sae-Lim et al. [38] performed impact analysis, which provides developers with ranked bad code smells with respect to their current context. Guggulothu et al. [39] suggested code smell order based on significant software metrics established using the feature selection process and studying the internal relation between code smells based on those software metrics. Sharma et al. [40] used multi-attributed machine learning models for determining the severity level of a reported bug in a cross-project context. Fontana et al. [14] used the intensity index for six familiar smells and found the most severe instances in the source code. They showed the role of intensity in examining and eliminating smells.

2.4 Comparison Studies Between Kotlin and Java

The domain of investigating Kotlin and its comparison with the Java language is less explored in the field of research. Bose et al. [15] compared Java and Kotlin in the android application domain with the result that Kotlin overwhelmed Java in many areas with respect to developers' point of view. Mateus et al. [16] compared the quality of android applications written in Kotlin to android applications written in Java language. It showed that the use of Kotlin increases the quality of code, which further optimizes the software quality. Coppola et al. [21] analyzed the adoption of Kotlin throughout the life of a set of android applications and studied whether they grant success to applications. The work showed that the choice of Kotlin was rapid and had minimal adoption cost parameters. Flauzino et al. [3] presented a comparative study of code smells in Java and Kotlin and found that Kotlin software code possesses fewer code smells than Java software code.

3 Experimental Setup

The liability to answer the research questions laid the foundation for this research work whose outcomes will surely help the developers and evaluators in enabling fewer resources, efforts, time, and rework cost. The entire workflow of this research is depicted in Fig. 1. First, the proposed methodology's deployment is applied to an appropriate and legitimate dataset of 30 open-sources software which are extracted from the GitHub repository and the code smells are chosen with the end goal that they are found common in Java as well as in Kotlin. Once the software is extracted, static code metrics evaluation is performed to gain the statistical insights of the software. Simultaneously, identification of code smells is done for Kotlin and further carried for Java using their respective tool and plugins.

Further, for applying various supervised machine learning algorithms, the labeled dataset is required. Therefore, Boolean identification is performed against the class containing considered code smell. The smelly instances indicating the presence of code smell is termed as true and the rest of the classes indicates the absence of code smell is termed as false. The dataset is then preprocessed and feature selection techniques are applied. The preprocessed dataset

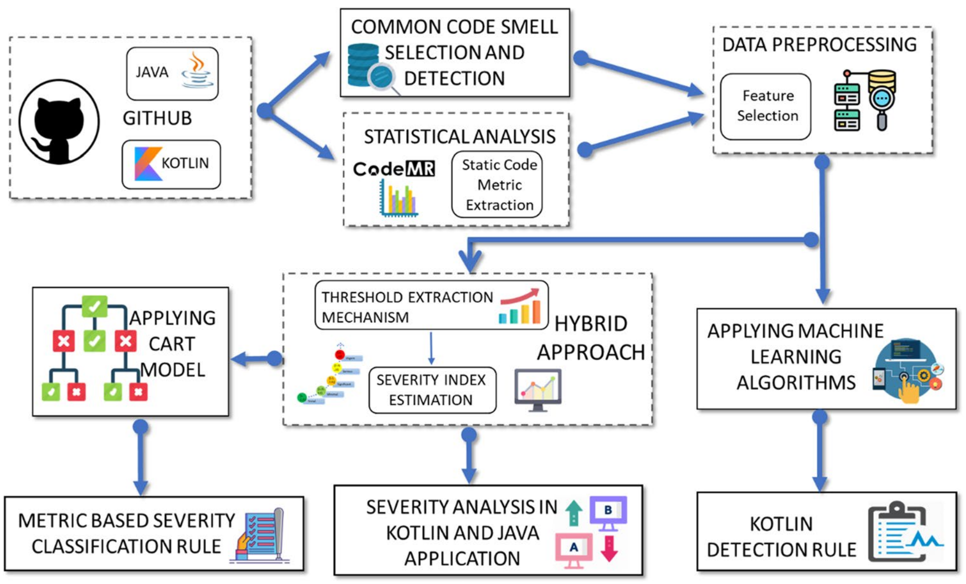

JAVA
GITHUB
KOTLIN
COMMON CODE SMELL SELECTION AND DETECTION
STATISTICAL ANALYSIS
CodeMR
Static Code Metric Extraction
DATA PREPROCESSING
Feature Selection
THRESHOLD EXTRACTION MECHANISM
HYBRID APPROACH
SEVERITY INDEX ESTIMATION
APPLYING CART MODEL
APPLYING MACHINE LEARNING ALGORITHMS
METRIC BASED SEVERITY CLASSIFICATION RULE
SEVERITY ANALYSIS IN KOTLIN AND JAVA APPLICATION
KOTLIN DETECTION RULE

Fig. 1 Overview of proposed research approach

Table 1 Summary of selected systems

Kotlin			Java		
Selected systems	LOC	No of classes	Selected systems	LOC	No of classes
anko	48,641	456	bitcoinj	76,008	393
Exposed	14,754	341	Bukkit	32,560	689
fuel	12,054	140	ChatApplicationJava	370	7
intellij-rainbow-brackets	3207	43	clojure	41,022	178
kotlin-koans	1782	78	easyexcel	15,407	322
kotlinpoet	20,198	121	hutool	78,729	1011
material-dialogs	6891	54	jadx	62,962	805
RxDownlaod	2856	62	jd-gui	15,864	221
RxKotlin	2463	19	joda-time	86,274	330
spek	5367	102	jsoup	20,208	110
ktx	16,578	235	junit4	31,026	447
ktor	92,196	798	Sentinel	63,862	857
leakcanary	19,220	185	Traccar	1711	1059
tivi	15,650	442	vert.x	120,320	762
CatchUp	15,462	339	shopizer	79,966	1116

of Kotlin has been evaluated by applying various machine learning algorithms. Then, the best machine learning algorithm is evaluated by Weka experimenter which is used for the formulation of human-readable code smell detection rules for Kotlin.

The proposed methodology is based on a hybrid approach for the fulfillment of the research goals. The hybrid approach aggregates two methods for extracting the thresholds to evaluate the severity index. The percentile method and the machine learning algorithm expectation–maximization are aggregated for the threshold extraction. The hybrid approach is performed for severity index evaluation in Kotlin and Java applications. Then, the methodology is automated using CART, machine learning algorithm. Finally, the *metric-based severity classification rules* are achieved from the CART algorithm. These rules are used for predicting the severity level of the code smell for prioritizing the refactoring process.

3.1 Dataset Criteria and Formulation

The experimentation is performed on 30 open-source software which comprises 15 Kotlin-based software and the rest 15 are Java-based software which are listed in Table 1. The implemented source codes of these systems are freely available and can be extracted and cloned from the global repository of open-source software, GitHub. The criteria followed for the selection of the systems depends on their acceptance

Table 2 Code smells and its description

Common code smells	Description
Complex method (CM)	Methods with high complexity, i.e., high McCabe's cyclomatic complexity (MCC)
Large class long method (LCLM)	Classes and methods which have more than one responsibility need to split up into smaller ones
Long parameter list (LPL)	Methods having several parameters, especially if most of them share the same datatype
String literal duplication (SLD)	Repeatedly typing out the same String literal across the codebase
Too many methods (TMM)	Classes with many methods should extract the functionality which belongs together in separate parts of the code

and reputation on the GitHub platform that can be evidenced by sorting the software on the basis of most stars. (The star marked software are those which are highly popular among the user, vastly used and the most relied software.) Additionally, another factor that is also considered is the chosen software for both the languages belong to different domains such as APIs (kotlinpoet, bukkit), Applications (catchup, tivi, chatApplication), and Frameworks (Exposed, ktor, Junit) which reflects diversity in the dataset. The dataset for Kotlin and Java are prepared individually, and the final dataset is accomplished and processed using several aspects, which are elaborated in Sects. 3.2 and 3.3

3.2 Code Smell Selection and Detection

This research work undertakes the five code smells which are found common in Kotlin as well as in Java. The detection process uses SonarKotlin[1] and Detekt[2] to investigate the presence of code smells in Kotlin, along with PMD[3] (popularly known as Programming Mistake Detector) in Java. The reason behind adopting these detection tools and plugins is that they are freely available and evaluates the same type of code smells for their detection. The code smells which are investigated in the course of research are described in Table 2.

The examination and selection criteria of code smell depend on certain aspects, first, on the type and effect of the particular code smell and second, the occurrence rate of the code smell while coding by practitioners. Besides these aspects, one more major benchmark considered is the existence of common code smells in Kotlin and Java. For instance, the smell cyclomatic complexity is found when the method possess high complexity, i.e., high McCabe's cyclomatic complexity (MCC) value. This smell is termed as complex method by Kotlin code smell detection tool '*Detekt*' whereas the same smell is termed as cyclomatic complexity by the Java code smell detection tool '*PMD.*' Similarly, the smell StringLiteralDuplication of Kotlin is termed as AvoidDuplicateLiterals in Java. Another factor that is also considered is that Kotlin is a new and emerging language, it is difficult to find out a satisfactory dataset with respect to a code smell; hence, the smell which lies in the same category are taken together. For instance, long method and large class are the smell which belongs to the same category, i.e., the smells occur when a class or a method grows too large tends to aggregate too many responsibilities and difficult to understand and therefore difficult to maintain; hence, these smells are taken together.

The presence of prevailing code smells opens up the path for the study on the prioritization for effective refactoring in Kotlin and Java applications and is illustrated along with advisors (detection tools and plugins) in Table 3. The Boolean identification of common code smells is performed manually at the coarse granularity level using these detection tools and plugins. This detection strategy aims to explore and inspect each code smell, which enhances the probability of finding the occurrences.

3.3 Statistical Code Analysis and Data Preprocessing

The statistical code analysis is a quantitative and qualitative measure representing the source code in the form of different static code metrics. The static code metrics which are obtained contain arbitrary entries that imitate several different properties and characteristics of the software with respect to the analyzed code. The study accounts for the static code metrics as a basis for further research, which is generated by analyzing individual systems. The generation

Table 3 Advisors (detection tools and plugins)

Code smells	Detection tools and plugin
CM	Detekt, PMD
LCLM	SonarKotlin, PMD
LPL	Detekt, Sonarkotlin, PMD
SLD	Detekt, Sonarkotlin, PMD
TMM	Detekt, PMD

[1] https://www.sonarsource.com/kotlin/.

[2] https://detekt.github.io/detekt/.

[3] https://pmd.github.io/latest/.

Table 4 Static code metrics from CodeMR

Categories			
Size	Complexity	Coupling	Cohesion
LOC	WMC	NOC	LCOM
NOF	DIT	CBO	LCAM
NOM	RFC	ATFD	LTCC
NOSF	SRFC		
NOSM	SI		
CMLOC			
NORM			

of static code metrics is achieved by a static code analysis tool called CodeMR.[4] The main reasons for choosing CodeMR are:

(a) CodeMR can easily be integrated with IDEs such as IntelliJ and Eclipse.
(b) CodeMR supports four programming languages—Java, Scala, Kotlin, and C++ and does not consider any other languages. Hence, the metric analysis process is not harmed or disturbed even if the project contains any other languages.
(c) CodeMR evaluates source code on distinct parameters like code complexity, cohesion, coupling, and size, as depicted in Table 4.

Further, data preprocessing is done to make the dataset ready for the implementation purpose. This involves cleaning, removing uncertainties, handling missing values, and class-level mapping for the presence of the code smells with static code metrics. This is done to remove all the irregularities from the dataset and develop a relationship between code smells and static code metrics. This step is regarded as a fundamental step in achieving the optimality of the dataset.

Henceforth, feature selection techniques are applied to select features from the dataset and reduce the dimensionality of data which may hamper the precision and accuracy of the final results. Further, the process of feature selection is accomplished with the help of two feature selectors, as follows:

1. Information Gain
2. Gain Ratio

The Info Gain algorithm evaluates the relatedness of an attribute with respect to the class by measuring Information Gain as follows [41]:

$$\mathrm{InfoGain}(C, A) = H(C) - H(C|A) \tag{1}$$

The Gain Ratio algorithm is an advancement in Info Gain, and it evaluates the relatedness of an attribute with respect to the class by measuring Gain Ratio as follows [42]:

$$\mathrm{GainRatio}(C, A) = (H(C) - H(C|A))/H(A) \tag{2}$$

where C represents the class and A represents the attribute, and the following equations are used to find the entropies of class, class | attribute and attribute, respectively:

$$H(C) = -\sum_{cEC} p(C) \log p(C) \tag{3}$$

$$H(C|A) = -\sum_{cEC} p(C|A) \log p(C|A) \tag{4}$$

$$H(A) = -\sum_{cEC} p(A) \log p(A) \tag{5}$$

They are evaluated with the help of data mining tool Weka[5] (Version 3.9.4), and the employment of such algorithms helps to adopt the best static code metrics in action, which preserves dataset processing time and human resources accordingly. Further implementation is explained in the upcoming Sect. 4.

3.4 Code Smell Severity Extraction Methodology

This research study employs a hybrid approach for the evaluation of the Kotlin and Java code smell severity for the considered software. The hybrid approach is formulated through the combination of two different approaches for the extraction of thresholds in the course of prioritization of code smell mentioned by Ferme and Fontana [14] and Alqmase et al. [28], i.e., *Percentile Method* and *Expectation–Maximization (EM) Method.* The approach is based on the static code metrics of the considered software which are obtained using the static code analysis tool CodeMR. Further, the hybrid approach has been applied for the computation of severity using the following steps:

1. Threshold Extraction Mechanism
   1.1 The initial step is the selection of static code metrics that can be considered for the extraction of severity index for respective code smell.
   1.2 The relevant static code metrics are obtained by considering all the metrics found in the detection rules obtained with the help of the JRip algorithm

[4] https://www.codemr.co.uk/.

[5] https://www.cs.waikato.ac.nz/ml/weka/.

explained in detail in RQ1. Subsequently, the software metrics for the Java language are also evaluated using the similar machine learning approach.

1.3 Percentile Method

1.3.1 The severity is extracted for the smelly instances; therefore, the hybrid approach works only on the smelly dataset. Each and every relevant static code metric is examined individually for the computation of thresholds over five different percentiles covering the entire range of smelly instances for a code smell.

1.3.2 The calculation of the initial percentile is performed by considering and evaluating the smallest smelly value of an instance for the static code metric. Furthermore, other percentiles are formed by observing four intervals and keeping them equally distributed so that they cover the entire range of code smell-affected classes.

1.3.3 The obtained percentiles for each attribute are then evaluated for their corresponding thresholds value by using a percentile rank function:

$$PR(i) = P(i) * (N + 1)/100 \tag{6}$$

where $PR(i)$ is the index number of the value at ith percentile, $P(i)$ is the percentile value, and N is the number of observations.

1.4 Expectation–Maximization (EM) Method: Similarly, thresholds are also computed with the application of unsupervised clustering machine learning algorithm—EM algorithm, which has been experimented with the help of data mining tool Weka (Version 3.9.4) for every relevant static code metrics.

2. Severity Index Estimation

2.1 The next step refers to the assignment of severity intensity corresponding to the static metric threshold values, separately for both methods. To ease in understanding, intensities are ranged in form of numerical values ranging from [21, 28] and further labeled for simple and effortless interpretation.

2.2 Besides this, the calculation process of intensities is also governed by the comparator operator $(>, <)$ of static code metrics, which is computed from JRip rules. Accordingly, the intensity may get higher when a particular static code metric is associated with operator '$>=$', whereas in the second case when the static code metric's operator is '$<=$', the intensities are reversed.

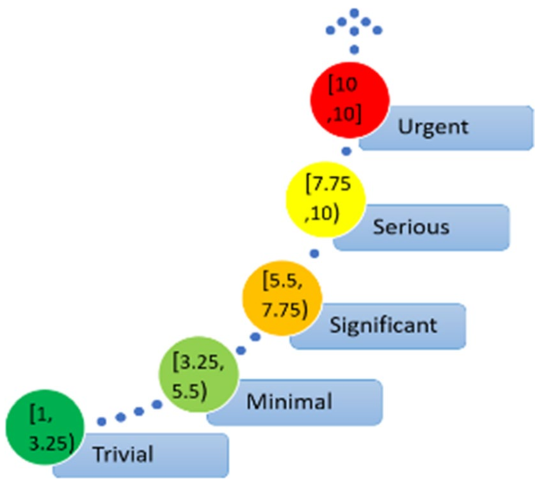


Fig. 2 Severity level with severity index

2.3 Next, the obtained intensities from both the methods (percentile and EM method) for every smelly class are aggregated to get the final combined intensities value termed as *hybrid intensity* for each relevant metrics of a code smell.

2.4 For obtaining single *severity intensity* at the class level, the *hybrid intensities* are again averaged which are obtained for each static code metrics. Further, these severity intensities are then labeled as illustrated in Fig. 2.

2.5 The last step is achieved by averaging the *severity intensity* values at the class level to obtain a single intensity value for a specific code smell which is finally termed as the *severity index* of the code smell.

This described process has been iteratively performed for every code smell and hence has been applied for identifying the *severity index* for both the considered languages which help to answer the RQ2 and RQ3.

4 Result and Discussion

The final dataset is qualified for further appraisement after the completion of the statistical code analysis and data preprocessing phase. The previously discussed phase increases the compatibility of the dataset to experiment and answer the research questions. The final dataset comprises of overall 17,075 classes of 15 considered Kotlin software and 41,535 classes of 15 considered Java software. Hence, the implementation and results of each research question are enlightened as follows:

Table 5 Weka experimenter corrected paired t-test result based on percent correct

Kotlin code smells	Jrip	RandomTree	PART	OneR
CM	98.06	97.5	98	98.06
LCLM	97.62	97.41	98	97.94
LPL	99.01	98.62	99.1	99.15
SLD	93.42	91.39	93.2	93.09
TMM	96.6	96.2	96.38	94.35

RQ1 Is there any method for developers to detect the code smells in Kotlin, and how accurately do the code smells follow it?

To answer the first question of the research, the application of machine learning has been experimented on the Kotlin dataset. The final dataset over which machine learning is applied contains only those attributes whose consequences are more significant according to the applied feature selection mechanism. To aid the developers, a semiautomated approach is established, which consists of the generation of human-readable detection rules to identify the code smell in Kotlin language. Hence, the detection rules are attained with the help of machine learning algorithm.

Machine Learning Approach

The exercise of machine learning is accomplished with the help of a tool called Weka, which is freely available for practicing data mining. To attain the detection rules, four different machine learning algorithms are compared, which are described as follows:

- *JRip*: It is a propositional rule learner, Repeated Incremental Pruning to Produce Error Reduction (RIPPER).
- *RandomTree*: Class for constructing a tree that considers K randomly chosen attributes at each node.
- *Part*: Class for generating a PART decision list using the C4.5 decision tree and makes the “best” leaf into a rule.
- *OneR*: Class uses a 1R classifier and minimum-error attribute for prediction, discretizing numeric attributes.

The best machine lea *rning* algorithm is evaluated with the help of Weka experimenter. The Weka experimenter is designed to facilitate the comparison of the predictive performance of algorithms based on the many different evaluation criteria that are available in Weka [43]. It uses the corrected paired *t*-test for the comparison between different machine learning algorithms. The corrected paired *t*-test is performed on the basis of percent correct at a significance level 0.05.

Results: The observation from the above comparison Table 5 proved that JRip algorithm results to be the best algorithm for the extraction of human-readable code smell detection rules for Kotlin language among all the considered four machine learning algorithms. The results provided by the JRip algorithm are also validated by performing 10-fold cross-validation technique and justified using the following performance measures [44, 45]:

Precision (Positive Predictive Value): It is measured as the fraction of instances predicted positives that are actually positive. It is also referred to as PPV.

$$\mathrm{Precision} = \mathrm{TP}/(\mathrm{TP} + \mathrm{FP}) \tag{7}$$

where TP is the true-positive rate and FP is the false-positive rate.

Recall: It measures what fraction of those instances that are actually positive were predicted positive.

$$\mathrm{Recall} = \mathrm{TP}/(\mathrm{TP} + \mathrm{FN}) \tag{8}$$

where TP is the true-positive rate and FN is the false-negative rate.

F-measure: It is a measure of the accuracy of a test and is defined as the weighted harmonic mean of the precision and recall of the test.

$$F - \mathrm{measure} = 2 * \mathrm{Precision} * \mathrm{Recall}/(\mathrm{Precision} + \mathrm{Recall}) \tag{9}$$

Accuracy: It is defined as the percentage of correctly classified instances by the considered classifier.

Mean Absolute Error (MAE): It is a linear score that measures the accuracy and average magnitude of errors for continuous variables.

ROC Area (Receiver Operating Characteristic): It is referred to as a plot of true-positive rate versus false-positive rate at distinct threshold points.

Kappa statistic: It identifies the random chance by measuring the relation between the observed accuracy with expected accuracy.

$$K = \Pr(a) - \Pr(e)/1 - \Pr(e) \tag{10}$$

where $\Pr(a)$ is observed agreement among the raters and $\Pr(e)$ is the hypothetical probability of the raters.

The best machine learning algorithm JRip is capable of detecting considered code smells with nearly 96% and 97% of overall precision and accuracy. The validated results obtained for each statistical measure are described in Table 6. The human-readable rules for the detection of code smell in Kotlin language detect the CM code smell when WMC is greater than or equal to 17, and LCAM is greater than or equal to 0.814, and CMLOC is greater than or equal to 138. Likewise, the human-readable detection rules for other considered code smells are illustrated in Table 7.

Moreover, the detection rules are also represented using cause-and-effect graph to show the relationships between underlined static code metrics with their values that appear

Table 6 Statistical analysis of code smells for JRip algorithm

Kotlin code smells	Precision	Recall	*F*-measure	Accuracy (%)	Mean absolute error	ROC area	Kappa statistic
CM	0.977	0.981	0.979	98.13	0.0258	0.636	0.2802
LCLM	0.975	0.979	0.977	97.86	0.0298	0.652	0.4056
LPL	0.985	0.991	0.988	99.06	0.0146	0.519	0.0561
SLD	0.928	0.937	0.931	93.70	0.1016	0.689	0.4476
TMM	0.967	0.963	0.965	96.34	0.0482	0.909	0.7635

Table 7 Human-readable code smell detection rules for Kotlin

Kotlin code smells	JRip detection rules
CM	(WMC > = 17) and (LCAM > = 0.814) and (CMLOC > = 138)
LCLM	(DIT < = 2) and (LOC > = 421)
LPL	(LOC > = 113) and (RFC < = 21) and (CMLOC > = 223)
SLD	(CMLOC > = 50) and (LCAM < = 0.75) and (DIT < = 1) and (RFC > = 62)
TMM	(NOM > = 11) and (CMLOC > = 34)

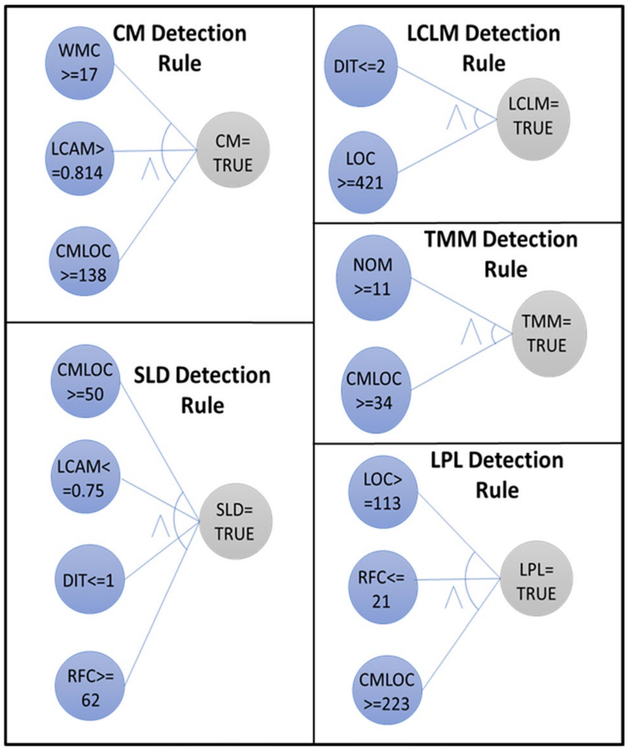


Fig. 3 Cause-and-effect graph for Kotlin detection rules

as causes and the code smells, which are seen as their effects. The rules contain the AND operation in between all the conditions or causes; hence, every condition or the causes should be true to get the smelly instance or the smell-affected class. The cause-and-effect graph of human-readable code smell detection rules is shown in Fig. 3. The

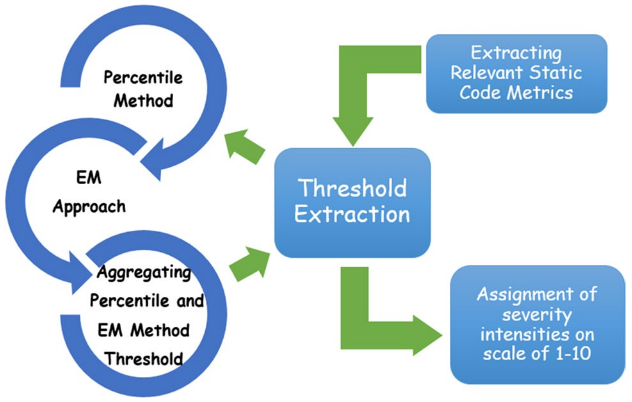

Percentile Method
EM Approach
Aggregating Percentile and EM Method Threshold
Threshold Extraction
Extracting Relevant Static Code Metrics
Assignment of severity intensities on scale of 1-10

Fig. 4 Severity extraction methodology

rules which are obtained are efficient and will surely benefit developers in evaluating code smells beforehand. The earlier identification of code smells will increase the optimality of the software and will decrease the possibilities of subsequent failures.

RQ2 Can a methodology be formulated for prioritization of code smell at coarse granularity level (class level) for Kotlin applications?

To answer the second question of the research, the calculation of static code metric thresholds and estimation of severity index is essential for prioritizing code smells to determine which code smell requires immediate removal from the software. The hybrid approach for severity Extraction is briefly illustrated in Fig. 4 and explained in detail in Sect. 3.4. The reasons for selecting the methods practiced in the hybrid approach for computing code smell severity have been stated below:

Percentile Method

The specific reasons for which the percentile method is chosen are:

(a) Its capability of covering all the observations which are present inside a particular percentile.
(b) It takes into account the statistical distribution of the metrics.
(c) It is fully repeatable as it allows the ranging of static code metrics values for identifying their thresholds at any level of granularity.

Expectation–Maximization (EM) Method

The next method, the EM method, is an efficient iterative unsupervised approach that generates the clusters of similar quality data for obtaining the metrics thresholds. The algorithm assigns a probability distribution to each instance, which indicates that it belongs to one of the clusters. The algorithm works effectively on the desired number of clusters. The file containing code smell-related static code metrics is input, and the output is the mean values that are estimated for extracting final thresholds.

Results: The harmfulness of code smells based on five severity levels are classified according to Fig. 2. The classes which are labeled as urgent resembles the most severe effect, and they require immediate refactoring action, whereas the classes which are labeled as trivial require least prioritized refactoring action. Further, it has been observed from the intensities obtained from both the approaching methods that produce almost similar results. The deflection noticed between both the intensity results is negligible and insignificant. It proves that both the methods are equally accountable for calculating the intensities for the static code metrics.

The estimation of the severity index is the major event for prioritization among code smells and the classes. The final computed severity index for considered code smells using the severity extraction methodology is displayed with the help of an interval plot in Fig. 5. These acquisitions will undoubtedly ensure that the severity index for prioritization of code smells and intensities for the prioritization of present classes in a code smell. Moreover, the established prioritization will help the software development, and maintenance team in refactoring the most severe code smells and provide the severe classes, which may harm and decreases the software's stability. The prioritization will also reduce the future costs and failures which are related to the software.

The interval plot provides information about the variation in the severity intensities of the code smells in Kotlin language. The plot is justified at 95% confidence level, and

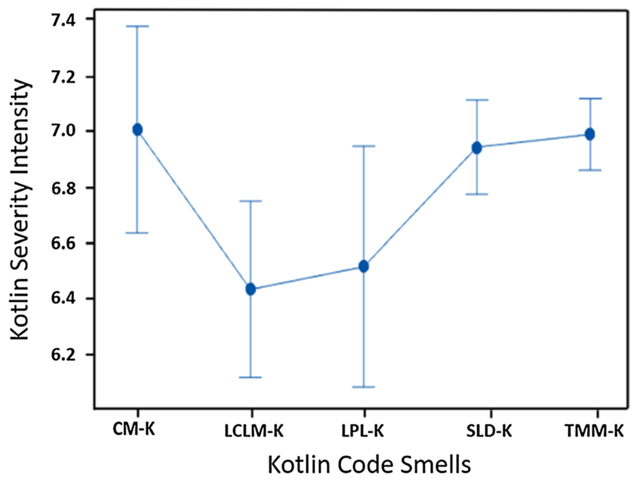


Fig. 5 Severity index for Kotlin code smells

the mean value for each code smell is regarded as its severity index. Further, the plot also provides a comparison of the amount of variation of severity intensities present in each code smell. The comparison among considered code smells proved that CM code smell had the highest severity index, and LCLM code smell has the lowest severity index. The remaining code smells TMM, SLD and LPL proved to furnish the second highest, third highest, and fourth highest severity index. In Kotlin, the severity index value obtained for code smells in descending order are 7.20, 7.18, 7.14, 6.71, and 6.62 approximately. This means that the developer should first take the CM code smell into account, and then the remaining code smells based on their computed severity index. This focus will highly contribute to achieving optimality in the software and further help the development team to prioritize their refactoring process. The accomplishment of such considerations will effectively save the time of the development team and will ensure high stability in the software when given a limited period.

RQ3 Can the common code smells be prioritized in Java applications for achieving effective refracting using hybrid approach?

Yes, the common code smells in Java application can be prioritized for effective refracting using the hybrid approach. The severity index for common code smells in Java has been computed similarly as elaborated in Sect. 3.4. Hence, this subsection is kept point to point for evaluation of the severity index of code smells in Java language using the hybrid approach.

Results: The result computed for Java severity analysis justifies that SLD code smell has the highest severity, and CM code smell has the lowest severity among the five considered code smells for Java language. Further, LCLM, TMM, and LPL code smells can be prioritized

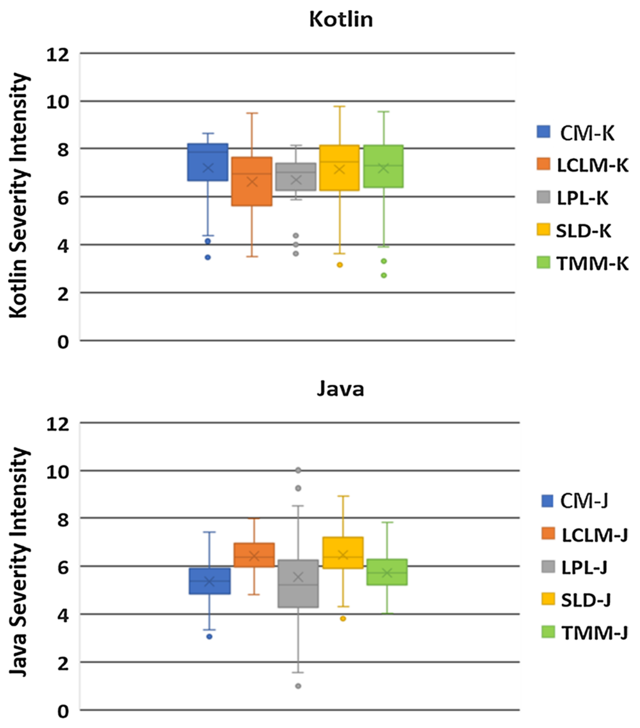
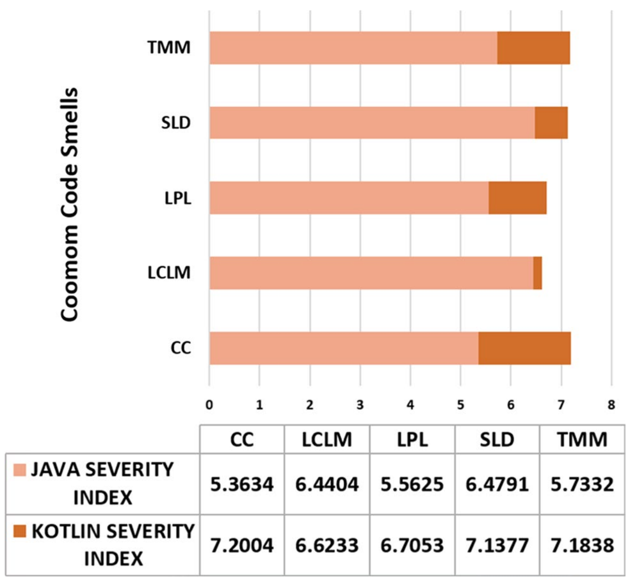


Fig. 6 Severity intensity distribution for Kotlin and Java



	CC	LCLM	LPL	SLD	TMM
■ JAVA SEVERITY INDEX	5.3634	6.4404	5.5625	6.4791	5.7332
■ KOTLIN SEVERITY INDEX	7.2004	6.6233	6.7053	7.1377	7.1838

Fig. 7 Overall analysis of severity index in Kotlin and Java for common code smell

based on calculated severity index values. Individually, the corresponding values for the severity of code smells are 6.48(SLD), 6.44(LCLM), 5.73(TMM), 5.56(LPL), and 5.36(CC) approximately. Therefore, the severity index values indicate the SLD code smell needs to be prioritized first, then LCLM at the second priority, then TMM and LPL and at last CC would be considered for the effective refactoring. Nevertheless, the dependency of severity index is solely on the static code metric distribution and their computed thresholds and corresponding severity intensities. The box plots signify the distribution of severity intensity of prevailing code smells in both the languages are illustrated in Fig. 6. In addition, a combined graphical representation of the analysis of severity index between Kotlin and Java is depicted in Fig. 7.

RQ4 Is it possible to semiautomate the hybrid approach for an in-depth classification to predict the severity level based on software code metrics?

To answer the fourth question of research, predictive analysis is needed to be thoroughly implemented on both the datasets. This question aims to predict the severity level for code smell affected classes with the help of static code metrics. In previous research questions 2 and 3, the entire computation process of severity is achieved manually, which consumes a massive amount of time and resources. To curtail these manifestations, semi-automation of the process for the assignment of severity level using the hybrid approach is required.

The semi-automation is achieved through the process of classification using decision trees, which is implemented with the support of the CART algorithm. Leo Breiman first proposed the CART model in 1984, and it is a nonparametric method that stands for classification and regression trees [46]. The objective of the highlighted algorithm is to build a model that predicts the value of a dependent or target variable with the help of independent or predictor variables. The applicability of classification rather than regression is more focused because the dependent variable is categorical here.

Implementation of CART Algorithm Using R

The CART algorithm's implementation is hardcoded for each code smells with the R language's companionship. The main reason for considering R is the unhindered access and reliability, which it offers to data scientists for performing several machine learning operations. The functioning of the CART algorithm for the construction of decision trees mainly follows a top-down approach. It places the most relevant variable on the root node and then matches the best split criteria for attaining better classification process. The best possible splitting criteria for each level are achieved with the help of the Gini index, which is defined as a splitting measure used by the CART algorithm to select optimal attribute candidates. The Gini index measures the probability of a randomly chosen attribute when it is incorrectly classified as follows [20]:

$$GI = 1 - \sum_{i=1}^{n} (p(i))^2 \tag{11}$$

Moreover, the experimentation of the CART algorithm is accomplished by installing three freely available libraries, which are named as rpart, rpart.control, plot, and caret. The consistency is maintained by dividing the dataset into training data, which accounts for 70% of the data, and the remaining 30% data is regarded as test data. The purpose of splitting the dataset is to ward off the overfitting and measure the model's performance. Hence, a very small part of the implementation for giving an insight is shown as follows:

Tree<-rpart(severitylevel~.,data=severityData,method="class")

Along with this, the CART algorithm requires a set of predictor variables that can predict the values of the target variable. In this case, the predictor variables are static code metrics that are present for each code smell, and the target variable is the severity level whose evaluation process is well explained earlier. Thenceforth, the CART algorithm is applied for the classification of severity level, which provides results in the form of decision trees that are retrieved and evaluated for producing final optimal results.

The inference from the decision trees provides classification rules which can be used for the prediction of severity levels for code smells in Kotlin and likewise for Java. These classification rules depend on the values of static code metrics and are called metrics-based severity classification rules. The final results are computed by choosing the optimal cp, which can inevitably acknowledge the genuineness of the used approach.

Results: While handling the CART algorithm, it is equally important to generate a tree of optimal size so that the tree might not overfit the dataset. The implementation part of CART is elaborated above, and further, the decision tree is generated in association with the CART algorithm. To achieve optimality, pruning is required and which is used to reduce the size of the tree by removing those undesirable branches which possess very less impact to classify instances. Moreover, pruning is done by selecting optimal cp, which is known as the complexity parameter. The cp is used to identify minimum improvement that is needed at each node in the model. The optimal cp is recognized from the graph, which is a plot between cp and relative error and whose process is explained ahead with examples. The adoption of optimal cp depends on two aspects: lower relative error and higher accuracy of the model. Hence, the point on the graph is identified as optimal cp when the relative error value is smaller than other cp. Moreover, if there is more than one cp whose relative error values are identical and smaller, in that case, cp with higher accuracy is chosen.

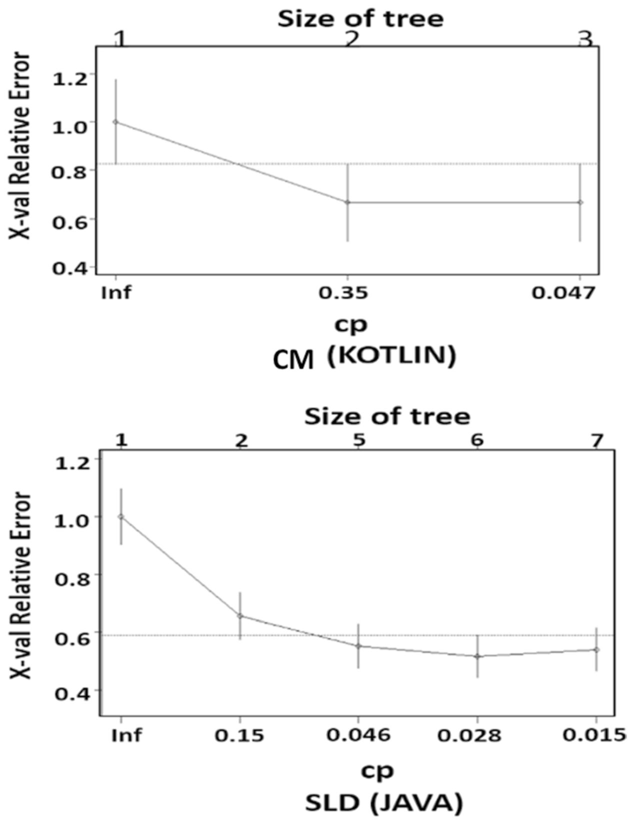


Fig. 8 Plot of CP vs relative error for complex method (CM) in Kotlin and string literal duplication (SLD) in Java

Therefore, optimal decision trees are generated for all the considered code smells in Kotlin and Java language except the LPL code smell. The main reason for LPL code smell could be the dataset as it contains fewer smelly classes than other code smells.

With a view to increase the understandability, one code smell each from both the languages is randomly chosen and explained. Next, the code smells chosen from Kotlin and Java are CM and SLD, respectively, whose entire procedure is described below. Initially, the input given is the metric-based severity levels of CM, and similarly, this is done for SLD. Further, the CART algorithm is implemented on these code smells individually. The plot of cp vs relative error is generated for code smells of respective languages in Fig. 8.

Figure 8 helps determine the complexity parameter for CM, and SLD code smells in Kotlin and Java, respectively. For CM in Kotlin, the smaller relative error is at 0.047 cp value, and for SLD in Java, the smaller relative error is at 0.028 cp value. Thenceforth, the optimal decision trees for both the smells at their respective cp values are shown distinctively in Figs. 9 and 10. Henceforth, a confusion matrix is also demonstrated for CM and SLD code smell in Table 8.

Similarly, these enactments are repeated for every code smell in Kotlin and Java with the help of optimal cp value.

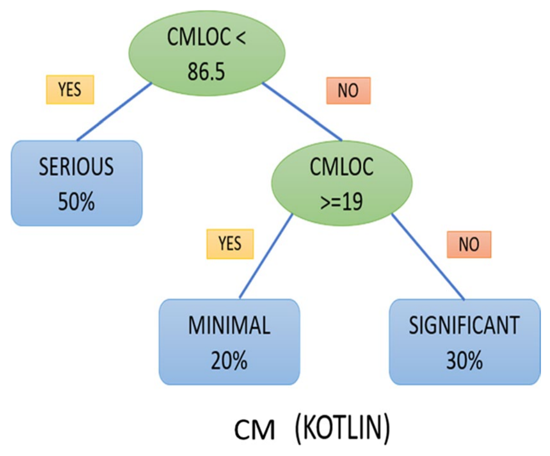


Fig. 9 Decision tree for complex method (CM) in Kotlin

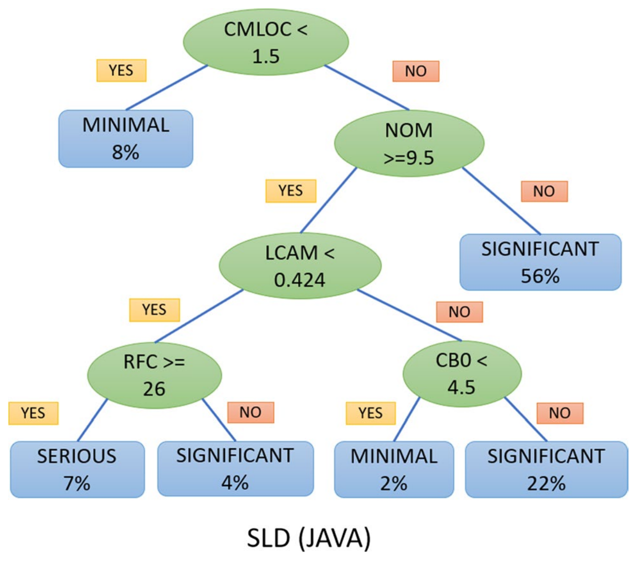


Fig. 10 Decision tree for string literal duplication (SLD) in Java

Hence, optimal decision trees are obtained for the considered code smells for both the domains. Further, the models are justified using different statistical metrics like accuracy, Kappa statistics, and AUC, which are already approached theoretically in this course of research work. The graphical representation of each statistical measure, which enhances model characteristics in areas of applicability, are depicted in Fig. 11.

The final interpretations from the decision trees are the metric-based severity classification rules for Kotlin and Java, which enhance the prediction of severity levels for estimating the severity of code smells. The metric-based severity classification rules extracted at their optimal cp value for Kotlin and Java are illustrated in Tables 9 and 10, respectively. It has been observed from the results that for severity levels serious, significant, and minimal for CM code smell in Kotlin is detected when CMLOC is greater than or equal to 86.5, CMLOC is lesser than 86.5 and greater or equal to 19 and CMLOC is lesser than 19, respectively. Thus, the metric-based severity classification rules for identification of severity levels of other code smells, which are considered in Kotlin and Java, can be referred from the tables constituting the rules.

The metric-based severity classification rules inference that the need to prioritize code smells can be established in a much more effortless way through the hybrid approach. The ease in the process of severity computation will positively help the developers to bring out the best in the software. Most importantly, developers will be able to focus more on highly severe code smells than the lower severe code smells while consuming minimum time, which can further help them to utilize their caliber in a resourceful manner.

5 Threats to Validity

This section discusses the threats to validity of the research work. The first source could be the experimentation which certainly induces errors in the formulation of the dataset. Since, Kotlin is an emergent language, it is hard to discover an acceptable dataset concerning a code smell; therefore, the Large Class and Long Method code smell falls in the same category are grouped together. Subsequently, the absence of adequate developer community and applications for Kotin language is also one of the reasons which could hamper the authenticity of the result. However, some applications which contain bad code are intentionally considered for preparing an effective dataset to identify the code snippets accurately which cause the code smell. The process of

Table 8 Confusion matrix for complex method (CM) in Kotlin and string literal duplication (SLD) in Java

CM of Kotlin				SLD of Java			
Prediction	Minimal	Serious	Significant	Prediction	Minimal	Serious	Significant
Minimal	3	0	0	Minimal	16	0	7
Serious	0	9	1	Serious	0	10	4
Significant	0	0	5	Significant	3	12	134

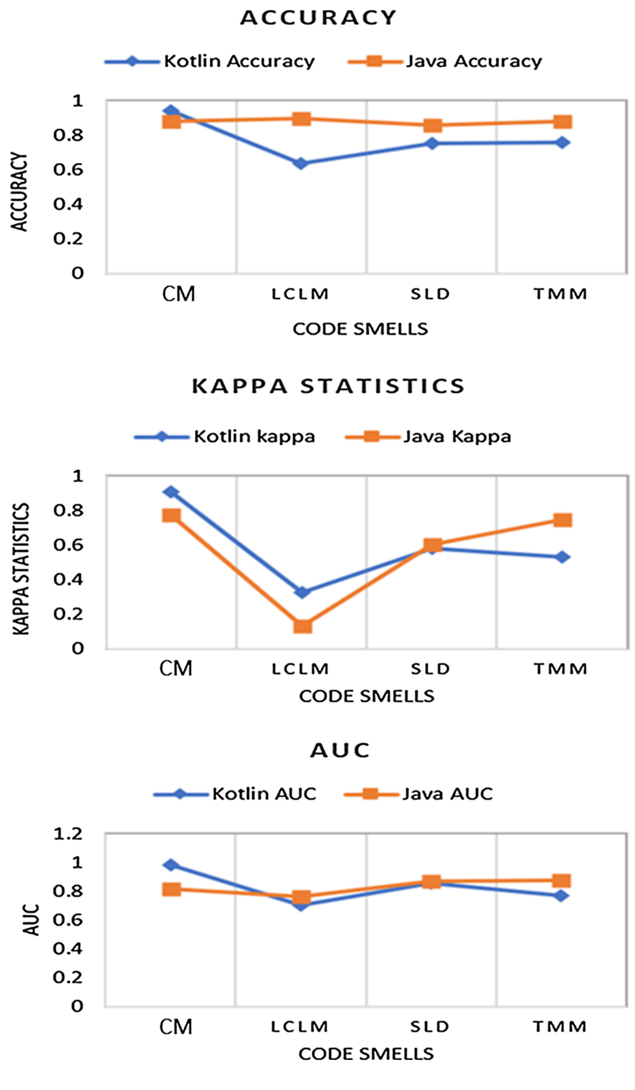


Fig. 11 Model statistical analysis of Kotlin and Java code smells

Boolean identification of code smells and aggregating the severity intensities to achieve the final severity indexes are performed manually whose aftereffects can be the introduction of human errors. In addition, the research work fairly compares five code smells, which are equivalent in Kotlin and Java systems.

Thus, the smelly instances possessed by the overall dataset are purely significant. However, the size of the smelly dataset could act as a barrier to the training of a machine learning model. As a result, the LPL code smell might be less compatible with the process of predicting the severity levels. Another factor is that the severity is calculated based on the static code metrics distribution. Severity is a customized process; it can be evaluated based on different factors other than metric distribution also and the result might deflect. Another potential threat could be the generalization of the obtained results. The results produced cannot be generalized for other languages because the dataset comprises 30 systems which are equally extracted from Kotlin and Java while ensuring proportionality for both the dataset. However, these results are specific to the respective language and the code smells that are considered, which arises in the source code.

6 Conclusion

The experiment is well established over five code smells, which are found disturbing the optimality of the software in Kotlin and Java and whose presence has always been a point of consideration of researchers are selected for evaluation. The common code smells which are studied are complex method, large class long method, long parameter list, string literal duplication, and too many methods, which are identified using different tools and plugins for respective

Table 9 Metric-based severity classification rules for Kotlin

Kotlin code smells	CP	Severity classification rules
CM	0.047	(CMLOC > = 86.5) = > SERIOUS
		(CMLOC < 86.5) and (CMLOC > = 19) = > SIGNIFICANT
		(CMLOC < 19) = > MINIMAL
LCLM	0.045	(LCAM > = 0.817) = > SERIOUS
		(LCAM < 0.817) and (CMLOC > = 13.5) = > SIGNIFICANT
		(LCAM < 0.817) and (CMLOC < 13.5) = > MINIMAL
SLD	0.018	(CMLOC > = 32.5) and (LCAM < 0.512) = > SERIOUS
		(CMLOC < 32.5) and (NOF < 1.5) = > SIGNIFICANT
		(CMLOC < 32.5) and (NOF > = 1.5) = > MINIMAL
TMM	0.033	(LCAM > = 0.6065) and (CMLOC > = 70) and (SRFC < 15) = > SERIOUS
		(LCAM < 0.6065) = > SIGNIFICANT

Table 10 Metric-based severity classification rules for Java

Java code smells	CP	Severity classification rules
CM	0.014	(RFC > = 7.5) and (NOSM < 0.5) = > MINIMAL
		(RFC < 7.5) and (DIT > = 0.5) = > SIGNIFICANT
LCLM	0.01	(NOM < 0.5) and (CBO < 3) and (LOC < 188.5) = > MINIMAL
		(NOM > = 0.5) = > SIGNIFICANT
SLD	0.028	(CMLOC < 1.5) = > MINIMAL
		(CMLOC > = 1.5) and (NOM < 9.5) = > SIGNIFICANT
		(CMLOC > = 1.5) and (NOM > = 9.5) and (LCAM < 0.424) and (RFC > = 26) = > SERIOUS
TMM	0.015	(NOSM < 0.5) and (LCAM > = 0.0715) and (LTCC < 0.931) and (LOC < 613.5) = > MINIMAL
		(NOSM > = 0.5) = > SIGNIFICANT

languages. To begin with, the dataset is preprocessed and experimented for various aspects of code smells arising in both the languages and their severe effects are analyzed through four research questions whose findings are targeted in the course of research work as follows:

Conclusion 1 Different machine learning algorithms were applied to the preprocessed dataset for semi-automating the code smell detection process in Kotlin language. The semiautomation is achieved in the form of human-readable code smell detection rules, which are practiced by best machine algorithms. The best algorithm out looked is the JRip algorithm with the help of the corrected paired *t*-test at a significance level 0.05, which is evaluated based on percent correct. Further, the human-readable code smell detection rules are also justified by 10-fold cross-validation technique and different statistical measures. The JRip algorithm perceives the best human-readable rules with an overall 96% precision and 97% accuracy for Kotlin code smell detection.

Conclusion 2 Two different approaches are employed for computing the thresholds of the static code metrics for the considered Kotlin code smells. Next, the computed thresholds are used to calculate the severity intensities for every smelly class using the hybrid approach. Further, the severity intensities are combined for the estimation of severity level and at last, the severity index is calculated for each Kotlin code smell. The final severity index obtained justifies that CM code smell had the highest severity index, and the LCLM code smell has the lowest severity index. The remaining code smells in decreasing order of their severity index are TMM, SLD, and LPL, respectively.

Conclusion 3 The severity index of code smells in Java is accomplished according to the hybrid approach as used for Kotlin. The results illustrate that SLD possesses the highest severity index whereas CC possesses the lowest severity index. Therefore, the SLD should be prioritized, afterward comes the LCLM then TMM, then LPL, and at last CC having the least severity index.

Conclusion 4 Rooted classification on the static code metrics is applied to reduce the human resources and limiting the time for the computation of the severity of code smells. The CART model is used for classification, and decision trees are generated for considered code smells. This process is applied to both the languages and optimal decision trees are evaluated on different statistical measures. After that, the inference from the decision trees generated at optimal cp is the metric-based severity classification rules, which can further be used for predicting the severity level for obtaining the code smell severity index.

In conclusion, the results from the research questions provide custom and effective conduct for the detection and prioritization of code smells in Kotlin and Java language. These conducts will surely help developers to optimize their identification and refactoring process for the code smells in both languages. Moreover, the maintenance phase of the software will charge fewer human efforts, time and rework costs, which will further increase the optimality of the software in a lesser time extent.

7 Future Work

The domain of code smells in Kotlin language requires more attention and research for exploring unseen actualities about design defects, which affect the characteristics and optimality of the systems. Moreover, the unconsidered code smells can be used for the generation of human-readable detection rules for Kotlin language. Similarly, the severity index can be estimated using the hybrid approach for the unconsidered code smells in Kotlin and likewise for Java. Moreover, the hybrid approach can be applied for computing the severity of code smells for other languages and domains so that the approach may get more generalized, and comparative analysis can be established between different languages and domains. The semiautomation can be achieved for computing severity index for unconsidered code smells and unconsidered languages.

Funding Not Applicable.

Availability of Data and Material The data is available on requirement basis.

Declarations

Conflict of interest The authors do not have any conflicts of interest.

References

1. Gupta, A.; Suri, B.; Bhat, V.: Android smells detection using ML algorithms with static code metrics. In: Batra, U.; Roy, N.; Panda, B. (Eds.) Data Science and Analytics: REDSET 2019: Communications in Computer and Information Science, Vol. 1229. Springer, Singapore (2020). https://doi.org/10.1007/978-981-15-5827-6_6
2. Parikh, G.: The Guide to Software Maintenance. Winthrop, Cambridge (1982)
3. Flauzino, M.; Veríssimo, J.; Terra, R.; Cirilo, E.; Durelli, V.H.S.; Durelli, R.S.: Are you still smelling it? In: Proceedings of the VII Brazilian Symposium on Software Components, Architectures, and Reuse on SBCARS '18. ACM Press (2018)
4. Tufano, M.; Palomba, F.; Bavota, G.; Oliveto, R.; Di Penta, M.; De Lucia, A.; Poshyvanyk, D.: When and why your code starts to smell bad. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering. IEEE (2015). https://doi.org/10.1109/icse.2015.59
5. Arcelli Fontana, F.; Zanoni, M.: Code smell severity classification using machine learning techniques. Knowl.-Based Syst. **128**, 43–58 (2017). https://doi.org/10.1016/j.knosys.2017.04.014
6. Fowler, M. (2018). Refactoring: Improving the Design of Existing Code. Addison-Wesley Professional.
7. Gupta, A.; Suri, B.; Kumar, V.; Misra, S.; Blažauskas, T.; Damaševičius, R.: Software code smell prediction model using shannon. Rényi and Tsallis Entropies. Entropy. **20**, 372 (2018). https://doi.org/10.3390/e20050372
8. Habchi, S.; Moha, N.; Rouvoy, R.: The rise of android code smells: Who is to Blame? In: 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR). IEEE (2019). https://doi.org/10.1109/msr.2019.00071
9. Belle, A.B.: Estimation and Prediction of technical debt: a proposal. Preprint http://arxiv.org/abs/1904.01001 (2019)
10. Cunningham, W.: The WyCash portfolio management system. SIGPLAN OOPS Mess. **4**, 29–30 (1993). https://doi.org/10.1145/157710.157715
11. Palomba, F.; Di Nucci, D.; Panichella, A.; Zaidman, A.; De Lucia, A.: On the impact of code smells on the energy consumption of mobile applications. Inf. Softw. Technol. **105**, 43–55 (2019). https://doi.org/10.1016/j.infsof.2018.08.004
12. Carette, A.; Younes, M.A.A.; Hecht, G.; Moha, N.; Rouvoy, R.: Investigating the energy impact of Android smells. In: 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER). IEEE (2017). https://doi.org/10.1109/saner.2017.7884614
13. Vidal, S.A.; Marcos, C.; Díaz-Pace, J.A.: An approach to prioritize code smells for refactoring. Autom. Softw. Eng. **23**, 501–532 (2016). https://doi.org/10.1007/s10515-014-0175-x
14. Fontana, F.A.; Ferme, V.; Zanoni, M.; Roveda, R.: Towards a prioritization of code debt: a code smell Intensity Index. In: 2015 IEEE 7th International Workshop on Managing Technical Debt (MTD). IEEE (2015). https://doi.org/10.1109/mtd.2015.7332620
15. Banerjee, M.; Bose, S.; Kundu, A.; Mukherjee, M.: A comparative study: Java vs kotlin programming in android application development. Int. J. Adv. Res. Comput. Sci. **9**(3), 41 (2018). https://doi.org/10.26483/ijarcs.v9i3.5978
16. Góis Mateus, B.; Martinez, M.: An empirical study on quality of Android applications written in Kotlin language. Empir. Softw. Eng. **24**, 3356–3393 (2019). https://doi.org/10.1007/s10664-019-09727-4
17. Gray, D.: Why does java remain so popular? https://blogs.oracle.com/oracleuniversity/why-does-javaremain-so-popular#:~:text=One%20of%20the%20most%20widely,%2C%20games%2C%20and%20numerical%20computing (2019). Accessed 12 April 2019
18. Shafirov, M.: Kotlin on Android Now official. https://blog.jetbrains.com/kotlin/2017/05/kotlin-on-android-now-official/ (2017). Accessed 17 May 2017
19. Sundin, E.: Perception and effects of implementing Kotlin in existing projects: a case study about language adoption (2018).
20. Schwermer, P.: Performance Evaluation of Kotlin and Java on Android Runtime (2018).
21. Coppola, R.; Ardito, L.; Torchiano, M.: Characterizing the transition to Kotlin of Android apps: a study on F-Droid, Play Store, and GitHub. In: Proceedings of the 3rd ACM SIGSOFT International Workshop on App Market Analytics - WAMA 2019. the 3rd ACM SIGSOFT International Workshop (2019). https://doi.org/10.1145/3340496.3342759
22. Sharma, T.; Spinellis, D.: A survey on software smells. J. Syst. Softw. **138**, 158–173 (2018). https://doi.org/10.1016/j.jss.2017.12.034
23. Chand, S.: Kotlin vs Java:Which is the best Android Development language?. https://medium.com/edureka/kotlin-vs-java-4f8653f38c04#:~:text=Kotlin%20has%20limited%20learning%20resources,circle%20is%20bigger%20than%20Kotlin (2019). Accessed 28 June 2019
24. Palomba, F.; Di Nucci, D.; Panichella, A.; Zaidman, A.; De Lucia, A.: Lightweight detection of Android-specific code smells: the aDoctor project. In: 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER). 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER) (2017). https://doi.org/10.1109/saner.2017.7884659

25. Kumar, N.A.; Krishna, K.H.; Manjula, R.: Challenges and best practices in mobile application development. Imp. J. Interdiscip. Res. **2**(12), 1607–1611 (2016)
26. Hecht, G.; Benomar, O.; Rouvoy, R.; Moha, N.; Duchien, L.: Tracking the software quality of android applications along their evolution (T). In: 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE (2015). https://doi.org/10.1109/ase.2015.46
27. Arcelli Fontana, F.; Ferme, V.; Zanoni, M.; Yamashita, A.: Automatic metric thresholds derivation for code smell detection. In: 2015 IEEE/ACM 6th International Workshop on Emerging Trends in Software Metrics. IEEE (2015). https://doi.org/10.1109/wetsom.2015.14
28. Alqmase, M.; Alshayeb, M.; Ghouti, L.: Threshold extraction framework for software metrics. J. Comput. Sci. Technol. **34**, 1063–1078 (2019). https://doi.org/10.1007/s11390-019-1960-6
29. Guggulothu, T.; Moiz, S.A.: Code smell detection using multilabel classification approach. Softw. Qual. J. **28**, 1063–1086 (2020). https://doi.org/10.1007/s11219-020-09498-y
30. Hozano, M.; Ferreira, H.; Silva, I.; Fonseca, B.; Costa, E.: Using developers' feedback to improve code smell detection. In: Proceedings of the 30th Annual ACM Symposium on Applied Computing - SAC '15. ACM Press (2015). https://doi.org/10.1145/2695664.2696059
31. Pecorelli, F.; Palomba, F.; Di Nucci, D.; De Lucia, A.: Comparing heuristic and machine learning approaches for metric-based code smell detection. In: 2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC). IEEE (2019). https://doi.org/10.1109/icpc.2019.00023
32. Sjoberg, D.I.K.; Yamashita, A.; Anda, B.C.D.; Mockus, A.; Dyba, T.: Quantifying the effect of code smells on maintenance effort. IEEE Trans. Softw. Eng. **39**, 1144–1156 (2013). https://doi.org/10.1109/tse.2012.89
33. Azeem, M.I.; Palomba, F.; Shi, L.; Wang, Q.: Machine learning techniques for code smell detection: a systematic literature review and meta-analysis. Inf. Softw. Technol. **108**, 115–138 (2019). https://doi.org/10.1016/j.infsof.2018.12.009
34. Saika, T.; Choi, E.; Yoshida, N.; Haruna, S.; Inoue, K.: Do developers focus on severe code smells? In: 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER). IEEE (2016). https://doi.org/10.1109/saner.2016.117
35. Kessentini, M.; Ouni, A.: Detecting android smells using multiobjective genetic programming. In: 2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft) (2017). https://doi.org/10.1109/mobilesoft.2017.29
36. Cruz, L.; Abreu, R.; Rouvignac, J.-N.: Leafactor: improving energy efficiency of android apps via automatic refactoring. In: 2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft). IEEE (2017). https://doi.org/10.1109/mobilesoft.2017.21
37. Zhang, M.; Baddoo, N.; Wernick, P.; Hall, T.: Prioritising refactoring using code bad smells. In: 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops. IEEE (2011). https://doi.org/10.1109/icstw.2011.69
38. Sae-Lim, N.; Hayashi, S.; Saeki, M.: Context-based approach to prioritize code smells for prefactoring. J. Softw. Evol. Proc. **30**, e1886 (2017). https://doi.org/10.1002/smr.1886
39. Guggulothu, T.; Moiz, S.A.: An approach to suggest code smell order for refactoring. In: Somani, A.; Ramakrishna, S.; Chaudhary, A.; Choudhary, C.; Agarwal, B. (Eds.) Emerging Technologies in Computer Engineering: Microservices in Big Data Analytics: ICETCE 2019: Communications in Computer and Information Science, Vol. 985. Springer, Singapore (2019). https://doi.org/10.1007/978-981-13-8300-7_21
40. Sharma, M.; Kumari, M.; Singh, R.K.; Singh, V.B.: Multiattribute based machine learning models for severity prediction in cross project context. In: Murgante, B., et al. (Eds.) Computational Science and Its Applications: ICCSA 2014. ICCSA 2014: Lecture Notes in Computer Science, Vol. 8583. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-09156-3_17
41. Pratiwi, A.I.: Adiwijaya: on the feature selection and classification based on information gain for document sentiment analysis. Appl. Comput. Intell. Soft Comput. **2018**, 1–5 (2018). https://doi.org/10.1155/2018/1407817
42. Gnanambal, S.; Thangaraj, M.; Meenatchi, V.T.; Gayathri, V.: Classification algorithms with attribute selection: an evaluation study using weka. Int. J. Adv. Netw. Appl. **9**(6), 3640–3644 (2018)
43. Hall, M.; Frank, E.; Holmes, G.; Pfahringer, B.; Reutemann, P.; Witten, I.H.: The WEKA data mining software: an update. ACM SIGKDD Explor. Newsl. **11**(1), 10–18 (2009). https://doi.org/10.1145/1656274.1656278
44. Mhawish, M.Y.; Gupta, M.: Generating code-smell prediction rules using decision tree algorithm and software metrics. ijcse. **7**, 41–48 (2019). https://doi.org/10.26438/ijcse/v7i5.4148
45. Gupta, A.; Suri, B.; Kumar, V.: Extracting rules for vulnerabilities detection with static metrics using machine learning. Int. J. Syst. Assur. Eng. Manag. **12**, 65–76 (2021). https://doi.org/10.1007/s13198-020-01036-0
46. Breiman, L.; Friedman, J.; Stone, C.J.; Olshen, R.A.: Classification and Regression Trees. CRC press. (1984)