Hindawi
Scientific Programming
Volume 2023, Article ID 2973250, 18 pages
https://doi.org/10.1155/2023/2973250

Research Article

Intelligent Mining of Association Rules Based on Nanopatterns for Code Smells Detection

D. Juliet Thessalonica,[1] **H. Khanna Nehemiah**,[1] **S. Sreejith**,[1] **and A. Kannan**[2]

[1] *Ramanujan Computing Centre, Anna University, Chennai 600025, India*
[2] *School of Computer Science and Engineering, Vellore Institute of Technology, Vellore, India*

Correspondence should be addressed to H. Khanna Nehemiah; nehemiah@annauniv.edu

Received 14 May 2022; Revised 17 December 2022; Accepted 31 January 2023; Published 13 April 2023

Academic Editor: Danilo Pianini

Copyright © 2023 D. Juliet Thessalonica et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

Software maintenance is an imperative step in software development. Code smells can arise as a result of poor design as well as frequent code changes due to changing needs. Early detection of code smells during software development can help with software maintenance. This work focuses on identifying code smells on Java software using nanopatterns. Nanopatterns are method-level code structures that reflect the presence of code smells. Nanopatterns are extracted using a command-line interface based on the ASM bytecode analysis. Class labels are extracted using three tools, namely inFusion, JDeodorant, and iPlasma. Rules are extracted from nanopatterns using the Apriori algorithm and mapped with the extracted class labels. Best rules are selected using the Border Collie Optimization (BCO) algorithm with the accuracy of the k-NN classifier as the fitness function. The selected rules are stored in the rule base to detect code smells. The objective is to detect a maximum number of code smells with a minimum number of rules. Experiments are carried out on Java software, namely jEdit, Nutch, Lucene, and Rhino. The proposed work detects code smells, namely data class, blob, spaghetti code, functional decomposition, and feature envy, with 98.78% accuracy for jEdit, 97.45% for Nutch, 95.58% for Lucene, and 96.34% for Rhino. The performance of the proposed work is competitive with other well-known methods of detecting code smells.

1. Introduction

Software maintenance is an important stage of the software development life cycle. The purpose of maintenance is to add functionality to the software system while maintaining its original functioning. These changes may appear as new requirements or previously planned but not implemented. Such changes affect either the algorithmic level or the specification level. Correcting faults, adapting to changing user requirements over time, updating hardware/software specifications, altering the components to remove undesirable side effects, and optimizing the code to execute more quickly are all the maintenance tasks necessary to ensure that the system continues to meet user needs. The imperativeness of software maintenance is to update and enhance the software after deployment to improve system performance. Maintenance begins after the software is built and made available to end users.

Maintenance includes strengthening existing capabilities, correcting errors, evaluating, and amending software to satisfy changing requirements [1]. Corrective, adaptive, perfective, and preventative maintenance are the four different categories of software maintenance. Corrective software maintenance addresses the bugs and faults in software applications that can influence the design, logic, and code. Adaptive software maintenance addresses software modifications that occur as a result of a change in the operating environment. Perfective software maintenance allows updating the software to improve its value according to user demands. Preventive software maintenance is a change in the software that detects and corrects latent faults before it becomes effective faults. These maintenance activities are hindered by code smells.

Code smells are distinct flaws related to improper structure, inappropriate object communication, and poor readability, all of which can negatively affect maintainability. Code smells are typically the result of deviations from object-oriented programming standards, which arise as a result of modifications made to software code to meet frequently changing customer requirements. Code smells are indicators of problems with the coding or design of the software. It can reduce the software's lifespan by making it difficult to maintain and evolve. According to Fowler [2], code smells are not inherently dangerous. They serve as a notification to the developers that the code may contain critical errors. Each smell has multiple symptoms that require a unique identifying mechanism [3]. The five code smells considered in this work are presented in Table 1. These code smells introduce excessive complexity and need more time to untangle the code.

Common approaches to detect code smells are search-based, metric-based, symptoms-based, visualization-based, probabilistic, cooperative-based, and manual [4]. The approaches to detect code smells fall into different categories, as shown in Table 2.

Reviewing related studies in similar areas reveals that the following issues remain challenging. The presence of code smells has been addressed by both researchers and practitioners, and they have proposed various methods, as outlined in Table 2, to detect them. However, none of these approaches addressed the uncertainty of the detection process. There is always a degree of uncertainty on whether a class in a program is a smell or not. Therefore, detection results should be reported with a probability corresponding to the degree of uncertainty of the detection process, as proposed by Khom et al. [5]. The results provided by the detection tool are usually different, making it challenging to compare them. Different tools may employ different metrics and thresholds to recognise a code smell depending on their detection rules. Therefore, machine learning techniques have been used to build a tool and assessed using 32 ML algorithms as proposed by Arcelli Fontana et al. [6]. Metric-based detection rules based on software metrics should be reported with the threshold values [7]. Detecting code smells and applying refactoring on one specific model can affect other related models. Refactoring suggested at the model level cannot be applied to the source code level. Therefore, a model-level refactoring method using a multiobjective evolutionary algorithm can be used to determine the optimal sequence of refactoring, as proposed by Mansoor et al. [8].

To find code smells, the approach described in this work uses association rules generated from nanopatterns. A pattern specifies structural connections between the classes through call or inheritance. In addition, it involves actions and interaction sequences among these classes. A pattern includes the classes, objects, and connections that make up the pattern's static structure, as well as behavioural pattern dynamics, such as messages that participants exchange. Both static and dynamic aspects are common in design patterns. Design patterns are reusable templates for structuring software Unified Modelling Language (UML) diagrams [9]. Nanopatterns are basic properties displayed by Java methods that can be traceable on a method or a procedure. They are traceable, which means they can be expressed as a simple formal condition on a Java method's attributes, types, names, and bodies [9]. Static analysis of Java bytecode identifies them. Bytecode is the compiled intermediate code that a Java virtual machine must translate to machine code. The bytecode instruction contained in Java class files are enriched in semantic information and indicate the execution process of the source code [10]. Although Gil and Maman [11] popularised the term "nanopattern" in 2005, their main focus was on micropatterns. Design patterns that are automatically recognised and used at a lower level of abstraction are connected to micropatterns. Host and Ostvold [12] later suggested a collection of Java method attributes, which Singer [13] referred to as nanopatterns. Table 3 provides an outline of the different types of patterns available in the software.

Jeremy Singer has created a command tool to find nanopatterns in bytecode class files based on the ASM bytecode analysis [14]. The detection tool iterates through a bytecode array looking for particular bytecode instructions to suggest particular nanopatterns. It is developed in Java and comprises 600 source lines of the code. A list of all methods and their associated nanopatterns is generated by the tool. An outline of the 17 fundamental nanopatterns grouped into four categories is presented in Table 4. The relationship between nanopatterns and code smells that already exists is the driving force for the choice of nanopatterns for code smell detection. ObjectCreator, FieldReader, FieldWriter, Looping, Exceptions, LocalWriter, and ArrayReader are nanopatterns that frequently appear in code smell procedures. Similar to this, SameName, NoReturn, Leaf, and StraightLine nanopatterns are prevalent in noncode smell approaches [15, 16].

In this work, a framework that uses rule generation and rule optimization to identify code smells has been proposed. Nanopatterns form the basis for the rules, which are extracted from the open-source software, namely jEdit, Nutch, Lucene, and Rhino, using a command-line interface. Datasets comprising methods and their corresponding nanopatterns without class labels are created from the open-source software. The class labels are the code smells present in each class or method, which are extracted using the tools, namely iPlasma, JDeodorant, and inFusion. Different tools generate different results depending on the computation of a specific set of software metrics ranging from standard object-oriented metrics to metrics defined in ad hoc ways for the purpose of code smell detection, which is difficult to interpret. As a result, the results of three tools considered in this work are intersected to generate an acceptable class label. Dataset containing nanopatterns without class labels are considered for association rule mining. The Apriori algorithm is used to generate frequent itemsets. Association rules are generated from the frequent itemsets and mapped to class labels. The Border Collie Optimization (BCO) algorithm is used to choose the best rules, and each rule is evaluated using the accuracy of the k-NN classifier as the fitness function. The optimal rules are stored in the rule base to detect code smells.

Table 1: Outline of code smells considered in this work.

Code smell	Outline
Data class	Class containing just the data, but cannot independently operate on the data
Blob	Class with more attributes and operations
Spaghetti code	Class containing methods with very large implementations invoke a single, multistage process flow and do not use an appropriate structuring mechanism
Functional decomposition	Class with the intent of performing a single function
Feature envy	Class attributes use another class attribute to perform computation rather than doing itself

Table 2: Outline of code smell detection approaches.

Approaches	Outline
Search-based	Solves optimization problems to find the best possible subset of solutions
Metric-based	Creates a rule based on metrics and respective thresholds
Symptom-based	Describes symptoms as class roles and structures that are transformed into detection algorithms
Visualization-based	Semiautomated process of visually representing data with metrics using visual metaphors
Probabilistic	Related to the degree of uncertainty of a class that indicates an occurrence of code smell
Cooperative-based	Improves performance and accuracy in detecting code smells by executing activities cooperatively
Manual	The human-centric process that requires a great human effort, extensive analysis, and interpretation effort from software maintainers to find design fragments that correspond to code smells

Table 3: Outline of software patterns.

Pattern	Outline
Design pattern	Package level nontraceable patterns that provide a repeatable solution to common software design problems
Micropattern	Class-level traceable patterns that provide formal conditions of the structure of a Java class
Nanopattern	Method-level traceable patterns that provide a group of reusable methods frequently used in Java class

Table 4: Outline of fundamental nanopatterns.

Category	Name	Outline
Calling	NoParams	Takes no arguments
	NoReturn	Returns void
	Recursive	Calls itself recursively
	SameName	Calls another method with the same name
	Leaf	Does not issue any method calls
Object-orientation	ObjectCreator	Creates new objects
	FieldReader	Reads field values from an object
	FieldWriter	Writes values to the field of an object
	TypeManipulator	Uses typecast or instance of operations
Control flow	StraightLine	No branches in a method body
	Looping	One or more control flow loops in a method body
	Exceptions	May throw an unhandled exception
Data flow	LocalReader	Read values of local variables on a stack frame
	LocalWriter	Writes values of local variables on a stack frame
	ArrayCreator	Creates a new array
	ArrayReader	Reads values from an array
	ArrayWriter	Writes values to an array

The remainder of this paper is organized as follows: Section 2 presents an overview of related works on nanopatterns and code smell detection. Section 3 outlines the proposed framework. Section 4 presents the results and their analysis. Finally, the conclusion and the scope for future work are presented in Section 5. Table 5 presents the abbreviations used in this work in alphabetical order.

2. Related Works

This section highlights the works carried out by other researchers related to nanopatterns and code smells.

Singer et al. [13] have constructed a command-line tool to extract the nanopatterns in Java class files based on the ASM bytecode analysis toolkit. With the aid of association

Table 5: Abbreviations Used.

Abbreviation	Phrase
ACO	Ant colony optimization
ABC	Artificial bee colony algorithm
AC	Associative classification
ACSM	Average concept similarity
APR	Active pruning rules
BA	Bat algorithm
BC	Brain class
BCO	Border collie optimization
BI-ADIPOK	Bi-level anti-pattern detection and identification using possibilistic optimized k-NNs
BM	Brain method
CBA	Classification based on associations
CBE	Class from the base example
CIM	Class from the initial model
CLOC	Changing lines of code
CSA	Cuckoo search algorithm
EGAPSO	Euclidean distance based genetic algorithm and particle swarm optimization
FA	Firefly algorithm
GA	Genetic algorithm
GP	Genetic programming
HCA	Hill climbing algorithm
HPSOM	Hybrid particle swarm optimization with mutation
LSA	Latent semantic analysis
MARC	Mining association rules from code
ML	Machine learning
PEA	Parallel evolutionary algorithm
PSO	Particle swarm optimization
UML	Unified modelling language

rule mining and Shannon entropy, the identified nanopatterns have been assessed. Two case studies have been presented: the first compares the relative object-oriented and diversity of two well-known Java benchmarking suites, SPECjvm98 and DaCapo; the second applies method-based clustering to the same Java benchmarking suites using nanopatterns as learning features. Both research studies have employed nanopatterns to produce concise summaries of Java methods, increasing their accuracy and efficiency. An extensive and diverse corpus of Java applications has been evaluated using a nanopattern identification technique to locate methods that exhibit nanopattern, yielding a 100% overall coverage score. The corpus contained 43,880 classes and 306,531 methods. Since the evaluation was given a perfect score, every approach had at least one nanopattern. The mean number of nanopatterns per method is typically 4.9.

Bruneton et al. [14] have developed an ASM toolkit, a Java manipulation tool to generate and manipulate Java classes. ASM employs the visitor design pattern in contrast to Java manipulation methods such as BCEL [17], SERP [18], and JOIE [19] to represent objects of classes. The visitor design pattern changes methods and the entire structure of a class without creating a new object for each bytecode instruction. ASM outperforms BCEL and SERP by 12 and 18 times, respectively.

Sultana et al. [16] have created a model to extract nanopatterns and predicted vulnerabilities for each method using an extraction tool based on the ASM bytecode analysis toolkit. Welch's t-test has been used to assess the relationship between vulnerabilities and nanopatterns, and the model has been trained using the nanopatterns as features. Utilizing nanopatterns as characteristics, three machine learning technique, namely naive Bayes, support vector machines, and logistic regression, has been used to predict vulnerability. The same techniques have been used with software metrics, and their effectiveness with nanopatterns has been compared. The results indicate that nanopatterns have a reduced false-negative rate of 21% compared to software metrics, which have a false-negative rate of 34.7% for classifying vulnerable methods.

Lu and Xu [20] have developed a toolkit named Kafer to facilitate the study of Java bytecode and the creation of software engineering tools for Java bytecode programs. The source code has been implemented in Java programming language, which runs on Windows and Sun Workstations. Users had a set of tools for slicing (Bslice), testing, maintaining (maintener), and gathering metrics on Java bytecode using this prototype implementation. The bytecode instructions in the source code can be changed when the maintener calls the Bslice to generate a slice and display it in the displayer. The quality of the software can be displayed using McCabe's metric. McCabe's cyclomatic complexity metric ensures that programs with high McCabe numbers (greater than 10) are difficult to understand and therefore have a higher probability of containing defects.

Mignon and de Azevedo da Rocha [21] have presented a mechanism to reduce the execution time of a Java program. The mechanism performed bytecode transformation, extracted nanopatterns, and measured execution time to

determine whether a method should be eliminated from the code. In addition, rules that specify which methods are not included in the execution process have been extracted utilizing the composition of nanopatterns. When specific methods have been eliminated, Java Grande Forum (JGF) benchmark suite and DaCapo have revealed a 23% improvement in JGF and a 43% in DaCapo programs.

Batarseh [22] has compared nanopatterns with design patterns and micropatterns. A collection of reusable methods known as nanopatterns has been widely applied in Java development. The system design has been described by design patterns. A class with no functions and only static members has been described by micropatterns. Three middle-sized systems, including a car navigation system using micropatterns, a flight black box simulator using design patterns, and a ticketing system using nanopatterns, have been the focus of experiments. The domain, size, and classes have been the same across all three systems. The amount of time required for development has been used to compare the three approaches. Patterns including Singleton, Bridge, and Façade have been used in flight black box simulators, while Joiner, DataManager, and Sink have been used in vehicle navigation systems, and ActionTriggered, Display, Initialize, Getter, Setter, Delete, Update, and Processing has been used in ticketing systems. Building the aforementioned systems required 20% for nanopatterns, 28% for design patterns, and 30% for micropatterns in total. The results demonstrate how nanopatterns for Java software minimize time, effort, and cost.

Saranya et al. [23] have proposed a hybrid approach that uses particle swarm optimization with mutation (HPSOM) for the detection of code smells. Fivecode smells: blob, spaghetti code, functional decomposition, data class, and feature envy have been investigated. The method has generated code smell detection rules by using a collection of software metrics and a set of code smell examples as input. Nineopen-source projects, including JFreeChart, GanttProject, Apache Ant 5.2, Apache Ant 7.0, Nutch, Log4j, Lucene, Xerces-J, and Rhino, have been used to test the methodology. The effectiveness of the approach has been compared to that of genetic algorithm (GA), particle swarm optimization (PSO), parallel evolutionary algorithm (PEA), and genetic programming (GP). This approach detects code smells with the highest precision of 97% on the GanttProject and a recall of 98% on the Log4J, respectively.

Saranya et al. [24] have addressed the problems with the rule-based approach and introduced a method for detecting smells utilizing the similarities between the software projects. The Euclidean distance-based genetic algorithm and particle swarm optimization (EGAPSO) method have been validated on two projects namely Log4J and GanttProject. The method has considered a class from the base example (CBE), software metrics, and a class from the initial model (CIM) as arguments and returned detected code smells from CIM. Five code smells namely, functional decomposition, blob, feature envy, spaghetti code, and data class, have been investigated. Recall of 96.3% and precision of 84.6% values have been found to be effective when compared to state-of-the-art techniques.

Tjortjis [25] has presented a method for Mining Association Rules from Code (MARC) to capture program structure, system knowledge, and assisting software management. Code parsing, rule mining, and rule grouping have been included in the methodology. Code has been initially parsed to add records, variables, and attributes to a database. The database has been used to extract association rules. The strength of the association rules linking the entities' items has been grouped together. COBOL programs have been used to test the methodology, and the results show that 72.75% of the created abstraction matched the expert mental model.

Rajab [26] has proposed a new Active Pruning Rules (APR) algorithm to increase predictive accuracy while minimizing rule redundancy. The algorithm has been implemented in two steps. Rules have been extracted using an Apriori algorithm based on minimum support and minimum confidence in the first step. The second step involves sorting the identified rules based on their support and confidence, and then tested on training data to identify those that can cover examples of data. Any rule that can cover at least one training data example has been added, and any rule that cannot cover has been removed from the classifier. APR produced 32 rules as opposed to 99 for the Classification Based on Associations (CBA) associative classifier. APR slightly increased predictive power while simultaneously reducing the size of the classifier. The experimental results reveal that APR outperforms other AC and rule-based classifiers.

Zang [27] has proposed an approach based on an immune optimization mechanism for optimizing associative classification rules. Both the rule searching and rule selection procedures take place at the same time as the immune cell population evolves. The optimization approach uses the clonal selection principle and immune memory mechanism to search for association rules. The approach has generated a wide range of local optimum solutions that have potential association classifier candidate rules. A new test instance can be classified using the classification model once it has been constructed. The proposed approach outperforms the standard associative classification (AC) algorithm in terms of runtime and accuracy, by achieving 92% for categorical and test datasets.

Mattiev and Kavsek [28] have proposed a new associative classifier that selects strong class association rules based on the overall coverage of the training set. The proposed classifier has the advantage of producing fewer rules on larger datasets while maintaining classification accuracy than existing classifiers. The frequent itemsets have been discovered using the Apriori algorithm. Once all frequent itemsets from a training dataset have been identified, class association rules are created. The rules have been arranged in decreasing order of confidence and support for the classification. The proposed method achieved an accuracy of 84.9% among all classification methods.

Awan and Shahzad [29] have proposed semisupervised associative classification using the ant colony optimization algorithm (ACO). The frequent patterns are identified by the algorithm. The associative classification rules have been

created by combining the class labels with frequent patterns. The antecedent is a pattern, and the consequent is a class of each rule. The confidence of each rule has been calculated, and confident rules are added to the rule list. The rule list has been sorted in descending order of confidence, followed by support. The constructed rules have been pruned to remove duplicate rules. The accuracy of the test set has been determined using the rule list. The experimental results show that the algorithm achieved an accuracy of 100%.

Pritam et al. [30] have evaluated code smells to predict changing classes in software. The authors proved that code smells, as compared to code metrics, can more accurately predict class changes. An open-source tool named Changing Lines of Code (CLOC) has been used to examine two versions of the same file to analyse the number of lines that have changed from the prior version of the software. After calculating the exact changes for each class, code smells have been detected using the Understand tool. Experiments have been conducted on AOI, Checkstyle, Freeplane, JKiwi, Joda, JStock, JText, LWJGL, ModBus, OpenGTS, OpenRocket, Quartz, Spring, and SubSonic. On the aforementioned software, six machine learning algorithms have been applied, namely naive Bayes, Multilayer Perceptron, Logitboost, Bagging, Random Forest, and Decision Tree. The results show that multilayer perceptron has been the most effective algorithm with a sensitivity of 70% and specificity of 67% to predict class changes using code smells.

Khari et al. [31] have implemented a testing tool with test suite generation and optimization functionality. Test data have been generated using black-box testing techniques namely boundary value testing, robustness testing, worst-case testing, robust worst-case testing, and random testing. The test data have been optimized using the artificial bee colony algorithm (ABC) and cuckoo search algorithm (CSA). The optimized test suite generates the actual output when applied to the software. By comparing actual and expected output, faults have been detected in the software. Experiments have been conducted on 10 sample programs for the optimization of the test suite. The results show that the average path coverage value for ABC over 10 programs is 90.3% while that for CSA is 75.4%.

Khari et al. [32] have proposed an approach for generating test suites and optimizing them during software testing. The approach focused on generating test suites using five Java programs. For each program, a control flow graph has been manually created to determine McCabe's cyclomatic complexity. The approach considered that the cyclomatic complexity of the program under test is equal to the number of test cases per test suite. The best test suites have been selected from the previously generated test suites using the six algorithms, namely hill climbing algorithm (HCA), bat algorithm (BA), cuckoo search algorithm (CS), firefly algorithm (FA), particle swarm optimization (PSO), and artificial bee colony algorithm (ABC). The results indicate the path coverage of ABC for the five programs is 97.8%, 99.8%, 50%, 97.92%, and 80%, respectively.

Son et al. [33] have explored all the software defect prediction literature available from 1995 to 2018 using a multistage process. A total of 156 studies have been selected in the first step, and inclusion-exclusion criteria have been applied to the resultant set to remove studies that do not match the objectives. The inclusion criteria have included the empirical study of software defect prediction using software metrics and studies that provide empirical analysis using statistical, search-based, and machine learning techniques. A quality analysis has been conducted to assess the relevance of studies. Meaningful information, namely authors, the title of publication, year of publication, datasets, and techniques used, have been extracted to perform data synthesis. Data synthesis accumulates the information collected from the data extraction process to build a response to research questions. The results are useful for the software engineering domain as well as for conducting empirical studies because step-by-step solutions are provided for questions raised in the article.

Zhang et al. [34] have suggested an approach DeleSmell to identify code smells using a deep learning model. A refactoring tool has been developed to convert a normal method into a brain method (BM) and a normal class into a brain class (BC). The iPlasma tool has been used to extract 24 structural metrics for BC code smell and 21 metrics for BM code smell. Latent semantic analysis (LSA) has been used to calculate the average concept similarity (ACSM) to measure the cohesion of the source code. The extracted features have been taken as input to the classifier. The classifier contains GRU-attention and CNN branch in parallel for feature selection. The selected features have been concatenated and sent to SVM for final classification. Experiments have been conducted on fop-core, JAdventure, MiniTwist, commons-lang, and redomar to detect BC and BM code smells. The results show that the approach achieved an average F-measure of 97.02% for BC and 98.22% for BM code smell detection.

Boutaib et al. [35] have developed a tool named Bilevel Anti-pattern Detection and Identification using Possibilistic Optimized k-NNs (BI-ADIPOK) that is capable of detecting and identifying code smells under certain and uncertain environments where uncertainty occurs at the level of class labels. Uncertainty factors are issued by human experts, which may result in a decrease in the quality of the result produced by detectors. BI-ADIPOK has used two levels of the code smell identification phase: the upper level generated a set of optimized PK-NNs parameters which were optimized using the lower level. The generated detectors have been trained on the chosen smell type to identify the specific code smells. Experiments have been carried out on Gantt-Project, ArgoUML, Xerces-J, JFreeChart, Ant-Apache, and Azureus to detect different code smells, namely blob, data class, feature envy, long method, duplicate code, long parameter list, spaghetti code, and functional decomposition. The results show that the recall precision curve varies between 0.902 and 0.932 for the uncertain environment and between 0.928 and 0.955 for a certain environment.

The following conclusions are drawn from a review of numerous studies in the literature. To begin, a command-line tool based on the ASM bytecode analysis toolkit can be used to extract nanopatterns. Second, there is a link between

nanopatterns and code smells, and they can be traced. This work proposes a framework for detecting code smells from the software system using rules generated from the nanopatterns. Initially, nanopatterns are extracted from the Java software using a command-line based on the ASM bytecode analysis toolkit. The code smells are extracted using the inFusion, JDeodorant, and iPlasma tools. These code smells serves as the class label for each method. Frequent itemsets are generated from the nanopatterns using the Apriori algorithm. Strong association rules are extracted from the frequent itemsets and mapped to the class label. The best rules are selected using the BCO algorithm and stored in the rule base for detecting code smells.

3. Materials and Methods

The proposed work for detecting code smells using nanopatterns includes subsystems, namely nanopattern extraction, class label extraction, rule extraction, mapping class labels with rules, and rule selection, as shown in Figure 1.

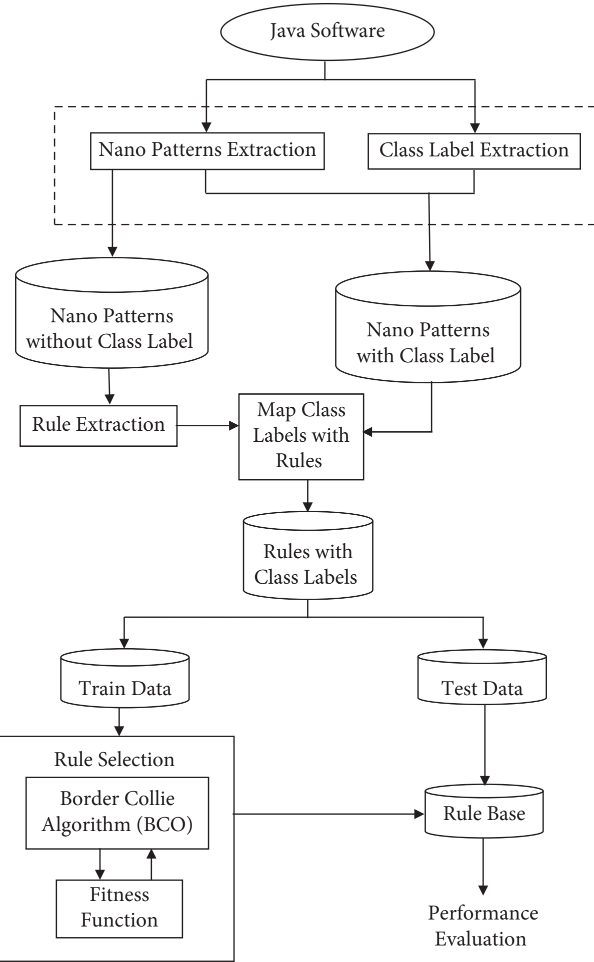


Figure 1: System framework.

3.1. Nanopattern Extraction. A command-line interface based on the ASM bytecode analysis is used to extract nanopatterns from Java bytecode class files. ASM is a tool for analysis and manipulation that enables the conversion of Java classes into binary formats. To extract nanopatterns, Java bytecode, an intermediate language between Java source code and assembly code is created. The tool searches a method bytecode array for certain bytecode instructions that reflect specific nanopatterns [12]. The tool for detecting nanopatterns can be downloaded as a jar file and run by typing *java -jar np.jar CLASSFILE*. The ASM bytecode manipulation library is used by the jar file.

The steps to extract nanopatterns from the jEdit software are outlined as follows:

Input: jEdit Java Source Code

Process:

Step 1: compile the Java source code and convert it into Java class file.

Step 2: extract the seventeen fundamental nanopatterns for each method using ASM-based command-line interface, as shown in Table 6.

Output: nanopatterns without the class label.

3.2. Class Label Extraction. Different software design measurement tools apply different methods for the detection of code smells. Code smell detection tools can be compared based on their performance. Comparing code smell detection tools with their results is a difficult task because different tools or plug-ins are built for different environments, code smells, and languages. Many code smells detection tools apply directly/indirectly different object-oriented source code metrics to detect a large number of code smells. Most tools have a textual output format, making it difficult to understand. The majority of code smell detection tools appear to be research prototypes or open-source projects. In comparison to commercial tools, research prototypes are less mature.

In this work, code smells are extracted from existing tools, namely inFusion, JDeodorant, and iPlasma tool. The reason behind choosing these tools is that they are built for a common environment and support the detection of code smells in Java source code. JDeodorant is an Eclipse plug-in that locates design flaws in software, also referred to as "bad smells," and fixes them by performing the proper refactorings. The software engineering group at the Department of Applied Informatics at the University of Macedonia in Thessaloniki, Greece, and the Software Refactoring Lab Department of Computer Science and Software Engineering at Concordia University in Canada have collaborated to create the tool. inFusion is engineered to put the development team in control over the architecture and design of the project. It aims to make quality control for software projects effective and feasible. iPlasma is an integrated environment for quality analysis of object-oriented software systems. It supports each stage of analysis, from model extraction through high-level metrics-based analysis, such as the detection of code smells. Three key benefits of iPlasma include its scalability, integration with additional analysis tools, and expansion of supported analysis. Different tools produce

Table 6: jEdit software-nanopattern without class label.

Method	FieldReader	FieldWriter	StraightLine	Looping	ArrayRreader	ArrayWriter	Exceptions
isKeyInTable	0	0	1	1	0	0	1
SetObject	0	0	0	0	1	0	1
isRightClickPopupEnabled	0	1	1	0	1	0	0
setRightClickPopupEnabled	1	0	0	0	0	0	0
isKnownExtension	0	0	1	0	1	1	0
handleTracks	0	0	0	0	0	0	1
translateDate	0	0	1	0	0	0	0
DoSearch	0	0	0	0	0	0	0
MapField	1	0	0	1	0	0	0
setTokenMarker	1	0	1	0	0	0	0

different conclusions on the same systems due to changes in the definitions and threshold values of source code metrics used for detecting code bad smells. As a result of this issue, the findings of three tools are intersected to provide acceptable code smells.

The steps to extract code smells from the jEdit software are outlined as follows:

Input: jEdit Java source code

Process:

Step 1: using inFusion, JDeodorant, and iPlasma tools to analyse the jEdit software

Step 2: select the jEdit software from each of the tool

Step 3: identify the flawed classes and methods

Step 4: identify five code smells, namely data class, blob, spaghetti code, functional decomposition, and feature envy in the flawed classes and methods

Step 5: intersect the code smells that the three tools have identified to obtain common code smells, as shown in Table 7

Output: code smells as class labels

3.3. Rule Extraction. Mining Association rules were introduced in 1993 by Agarwal [36]. In this work, association rules are generated from the dataset containing nanopatterns using the Apriori algorithm [37]. All itemsets that have a minimum of 4% support have been generated as frequent itemsets. The association rule is an implication expression of the form, $X \longrightarrow Y$ where X and Y are disjoint itemsets, i.e., $X \cap Y = \varnothing$. The strength of an association rule can be measured in terms of its confidence. Support value is the frequency of the occurrence of X and Y or $P(X \cup Y)$. Confidence is the conditional probability of X and Y or $P(Y|X)$. All the rules that satisfy prespecified minimum confidence can be generated from the nanopatterns, as shown in Table 8.

The steps taken in the Apriori algorithm are outlined as follows:

Input: nanopatterns without a class label as a dataset

Process:

Step 1:find all 1-itemsets and frequent 1-itemsets candidates from the dataset

Step 2: find all frequent 2 and 3-itemsets with minimum support of 4%, as shown in Figure 2

Step 3: generate a hash structure for storing k-itemsets

Step 4: generate association rules from the frequent k-itemsets with a confidence greater than 50%, as shown in Table 9

Output: association rules

3.4. Mapping Class Labels with Rules. After discovering the possible association rules they need to be mapped with the class labels. The class labels are the code smells detected from three open-source tools, as discussed in Section 3.2. Table 10 shows nanopatterns with class labels. The extracted nanopatterns and class labels are then associated by method mapping.

The steps taken in the method mapping are outlined as follows:

Input: nanopatterns for each method, class labels for each method

Process:

Step 1: assign key values for each method ($m1$, $m2$, and $m3$) in the class label extraction, as class labels blob (BL), feature envy (FE), and data class (DC) in map1

Step 2: assign key values for each method in the nanopattern extraction, as nanopatterns ($n1$ to $n17$) in map2

Step 3: check for equality by verifying that the methods in map1 are equal to the methods in map2

Step 4: if map1equals map2, then copy the corresponding class label in map1 to map2

Output: nanopattern with class labels

Let $A = \{a_1, a_2, \ldots, a_m\}$ be a finite set of all attributes in dataset. $C = \{c_1, c_2, \ldots, c_n\}$ is a set of classes, $g(x)$ is a set of transactions containing ruleItem x, and $|g(x)|$ is the number of transactions containing x.

The confidence of ruleItem <ruleItem, c_i> is the ratio of the number of transactions that contain the ruleItem in class c_i and the number of transactions containing the ruleItem, as in equation (1).

Table 7: Code smells detected using inFusion, JDeodorant, and iPlasma.

Code smell	inFusion				JDeodorant				iPlasma			
	jEdit	Nutch	Lucene	Rhino	jEdit	Nutch	Lucene	Rhino	jEdit	Nutch	Lucene	Rhino
Blob	3	17	5	15	8	25	2	11	5	11	3	13
Feature envy	8	5	7	11	10	10	3	7	5	4	5	9
Functional decomposition	4	0	2	7	0	0	0	1	2	0	1	7
Spaghetti code	2	0	5	6	0	0	3	2	2	1	2	5
Data class	7	21	13	28	5	18	15	24	4	22	11	19
Total	24	43	32	67	23	53	23	45	18	38	22	53

Table 8: Nanopatterns without class label.

TID	FieldReader	FieldWriter	StraightLine	Looping	ArrayReader	ArrayWriter	Exceptions	ObjCreator	LocalReader	LocalWriter
1	1	0	1	1	0	1	1	1	1	0
2	0	0	0	0	1	0	1	1	0	0
3	0	1	1	0	1	0	0	0	1	1
4	1	0	0	0	0	0	0	1	1	1
5	0	0	1	0	1	1	0	0	0	0
6	0	0	0	0	0	0	1	1	1	0
7	0	1	1	0	0	0	0	0	0	1
8	0	1	0	0	0	1	0	1	1	0
9	1	0	0	1	0	1	0	1	0	1
10	1	1	1	0	0	0	0	1	1	1

$$\text{Conf}\left(\langle \text{ruleitem}, c_i\rangle\right) = \frac{\left|g\left(\langle \text{ruleitem}, c_i\rangle\right)\right|}{\left|g\left(\text{ruleitem}\right)\right|} \times 100. \quad (1)$$

Considering the first ruleItem FieldR eade r = > ObjCreator from Table 9 that occurs in the transaction IDs 1, 4, 9, and 10 as given in Table 10. It is denoted as $g(\langle \text{FieldReader} => \text{ObjCreator}\rangle) = \{1, 4, 9, 10\}$, class blob occurs in the transaction IDs 1, 4, and 10 and denoted as $g(\text{Blob}) = \{1, 4, 10\}$, while feature envy occurs in the transaction IDs 2 and 6 and denoted as $g(\text{Feature Envy}) = \{2, 6\}$, data class occurs in the transaction IDs 5 and 8 and denoted as $g(\text{Data Class}) = \{5, 8\}$, functional decomposition occurs in transaction IDs 3 and 7 and denoted as $g(\text{Functional Decomposition}) = \{3, 7\}$, and spaghetti code occurs in transaction ID 9 and denoted as $g(\text{Spaghetti Code}) = \{9\}$.

The transaction IDs containing $\langle \text{FieldReader} => \text{ObjCreator}\rangle \longrightarrow \text{Spaghetti Code}$ are $(\langle \text{FieldReader} => \text{ObjCreator}\rangle) \cap g(\text{Spaghetti Code}) = \{1, 4, 9, 10\} \cap \{9\} = \{9\}$, so the support for $\langle \text{FieldReader} => \text{ObjCreator}\rangle \longrightarrow \text{Spaghetti Code}$ is 1. Hence, this rule is not mapped with the class label spaghetti code. The transaction IDs containing $\langle \text{FieldReader} => \text{ObjCreator}\rangle \longrightarrow \text{Blob}$ are $(\langle \text{FieldReader} => \text{ObjCreator}\rangle) \cap g(\text{Blob}) = \{1, 4, 9, 10\} \cap \{1, 4, 10\} = \{1, 4, 10\}$, so the support for $\langle \text{FieldReader} => \text{ObjCreator}\rangle \longrightarrow \text{Blob}$ is 3. The confidence of $\langle \text{FieldReader} => \text{ObjCreator}\rangle \longrightarrow \text{Blob}$ will be calculated for the rule having greater support as $(|g(1, 4, 10)| / |g(1, 4, 9, 10)|) \times 100 = (3/4) \times 100 = 75\%$. Since this rule $\langle \text{FieldReader} => \text{ObjCreator}\rangle \longrightarrow \text{Blob}$ has greater support and confidence, it is mapped with the class label blob.

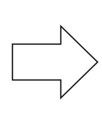
Similarly, the support and confidence for the other ruleItems to the class labels are calculated. The selected rules with class label are stored in the dataset. The dataset is then split into training and testing set in the ratio 80 : 20.

3.5. Rule Selection. A large number of rules with minimum support and confidence are generated in the rule extraction phase. The best rules are selected using the BCO algorithm. The algorithm requires binary transformation, in which the dataset is transformed into binary data in the form of 1 s and 0 s, with 1 indicating the presence of an item and 0 indicating its absence. Each individual is represented as a separate rule. Let N be the total number of items in the dataset. Each ruleItem is represented by two bits, each of which can be either 0 or 1, as shown in Table 11. The first bit has a value of 1 if the item is present in the rule, and 0 otherwise. The second bit indicates whether the item is in the antecedent (takes 1) or consequent (takes 0). A ruleItem (RI) can be represented in either of four possible combinations. The attributes should be restricted to the antecedent side of the rule, and the class label should be restricted to the consequent side of the rule. As a result, in individual encoding, the Bit 2 value of the class label is 0, and the Bit 2 value of the ruleItem is 1.

The rule selection uses the Border Collie bio-inspired algorithm to select the optimal rule containing the code smell methods. The accuracy of the k-NN classifier is used as

Iteration 1

Items	Support
FieldReader	0.05
FieldWriter	0.04
Straightline	0.05
Looping	0.02
ArrayReader	0.03
ArrayWriter	00.4
Exceptions	0.03
ObjCreator	0.07
LocalReader	0.06
LocalWriter	0.05



Frequent 1-itemsets

Items	Support
FieldReader	0.05
FieldWriter	0.04
Straightline	0.05
ArrayWriter	0.04
ObjCreator	0.07
LocalReader	0.06
LocalWriter	0.05

Iteration 2

Items	Support
(FieldReader, FieldWriter)	0.01
(FieldReader, Straightline)	0.01
(FieldReader, ArrayWriter)	0.02
(FieldReader, ObjCreator)	0.04
(FieldReader, LocalReader)	0.03
(FieldReader, LocalWriter)	0.03
(FieldWriter, Straightline)	0.03
(FieldWriter, ArrayWriter)	0.02
(FieldWriter, ObjCreator)	0.02
(FieldWriter, LocalReader)	0.03
(FieldWriter, LocalWriter)	0.03
(Straightline, ArrayWriter)	0.02
(Straightline, ObjCreator)	0.02
(Straightline, LocalReader)	0.03
(Straightline, LocalWriter)	0.03
(ArrayWriter, ObjCreator)	0.03
(ArrayWriter, LocalReader)	0.02
(ArrayWriter, LocalWrter)	0.01
(ObjCreator, LocalReader)	0.05
(ObjCreator, LocalWrter)	0.03
(LocalReader, LocalWrter)	0.03

Frequent 2-itemsets

Items	Support
(FieldReader, ObjCreator)	0.04
(ObjCreator, LocalReader)	0.05

Iteration 3

Items	Support
(FieldReader, ObjCreator, LocalReader)	0.04

Frequent 3-itemsets

Items	Support
(FieldReader, ObjCreator, LocalReader)	0.04

Figure 2: Frequent 1, 2, and 3 itemsets.

Table 9: Possible association rules.

Itemsets	Confidence
FieldReader ≥ ObjCreator	0.8
ObjCreator ≥ FieldReader	0.57
ObjCreator ≥ LocalReader	0.71
LocalReader ≥ ObjCreator	0.83
(FieldReader, ObjCreator) ≥ LocalReader	1.00
(FieldReader, LocalReader) ≥ ObjCreator	1.33
(LocalReader, ObjCreator) ≥ FieldReader	1.33

the fitness function for the Border Collie algorithm. After the training, the larger portions (i.e., 80%) of the rules are divided further into the ratio of 80 : 20, 80% is used for training the rules, and 20% is used for testing the constructed k-NN classifier. The k-NN stores the training data and uses it to classify new observations based on the value of the k-closest stored points. The k-value plays a significant role in determining which rules are appropriate to use to maximize the k-NN classifier's accuracy. Based on the optimum fitness value the rules are selected. At this optimal k-value, the classifier's accuracy improves as well. The parameter settings of the Border Collie Optimization algorithm are presented in Table 12.

Border Collies are an affectionate, smart, and energetic breed of dogs. They are extremely intelligent, athletic, and can be easily trained. These dogs are usually healthy and active, having a normal life span of about 12 to 15 years. It can be said that watching a Border Collie herd sheep is like watching a master craftsman at work. Herding is an inherent ability they are born with. Even when a puppy is introduced to the herd for the first time, they demonstrate immense control over the sheep. Border Collies follow three herding techniques, namely gathering, stalking, and eyeing. In Border Collie Optimization (BCO), a population of three dogs, and sheep is considered. A group consisting of three dogs and sheep is visualized while initiating the algorithm.

Table 10: Nanopatterns with class labels.

TID	FieldReader	FieldWriter	StraightLine	Looping	ArrayReader	ArrayWriter	Exceptions	ObjCreator	LocalReader	LocalWriter	Class label
1	1	0	1	1	0	1	1	1	1	0	BL
2	0	0	0	0	1	0	1	1	0	0	FE
3	0	1	1	0	1	0	0	0	1	1	FD
4	1	0	0	0	0	0	0	1	1	1	BL
5	0	0	1	0	1	1	0	0	0	0	DC
6	0	0	0	0	1	0	1	1	1	0	FE
7	0	1	1	0	0	0	0	0	0	1	FD
8	0	1	0	0	1	1	0	1	1	0	DC
9	1	0	0	1	0	1	0	1	0	1	SC
10	1	1	1	0	0	0	0	1	1	1	BL

Note. BL: blob, FE-feature envy, FD: functional decomposition, SC: spaghetti code, and DC: data class.

Table 11: Rule representation in BCO.

RI_1		RI_2		RI_3		...	Class label	
Bit 1	Bit 2	Bit 1	Bit 2	Bit 1	Bit 2	...	Bit 1	Bit 2

Table 12: Parameter settings for BCO.

Parameter	Value
Initial number of individuals (N)	$N = 30$ (3 dogs and $N - 3$ sheep)
Velocity of each individual	0
Time of each individual	Random number between 1 and 30
Acceleration of each individual	1
Maximum no. of iterations	100

The sheep go out for grazing in different directions and the dogs are responsible for bringing them back to the farm [38].

The steps involved in the BCO algorithm for rule selection are presented as follows:

Input: Rules with class labels—Training Data

Process:

Step 1: Initialize the population of N Border Collies (solution) and sheep at random. Each Border Collie and sheep is a possible solution (rule) that contains nanopatterns of length “n.” If the corresponding nanopattern is selected for the rule, it is represented as “1”, else as “0”.

Step 2: Calculate the fitness function of each Border Collies and sheep (rule) using the accuracy of the k-NN classifier. Each rule is evaluated with the classification accuracy as given in equation (2).

$$\text{Classification accurcay} = \frac{\text{Number of instances classified correctly}}{\text{Total number of instances classified}}. \quad (2)$$

(i) The Border Collie, a lead dog with a greater level of fitness (classification accuracy), has been recognised as the most accurate solution. The dogs—lead dog, right dog, and left dog—and sheep—gathered sheep, stalked sheep, and eyed sheep—are named after their diminishing fitness levels.

(ii) The individual with the best fitness (fit_f) is designated as the lead dog, in every iteration and is responsible for mainly gathering.

(iii) Individuals with the 2nd and 3rd best fitness values are chosen as left and right dogs. A tournament selection method is applied to choose left and right dogs. These dogs mainly participate in the stalking and eyeing of the herd. Their fitness values are referred to as (fit_{le}) and (fit_{ri}), respectively.

Step 3: The optimum solution is the dogs to lead the sheep to the farm. They travel from one point in the field to the farm. The distance covered and direction of the sheep and dogs are controlled by velocity, acceleration, and time.

Step 3a: The velocity of dogs is calculated using equations (3) to (5).

$$V_f(t+1) = \sqrt{V_f(t)^2 + 2 \times \text{Acc}_f(t) \times \text{Pop}_f(t)}, \quad (3)$$

$$V_{ri}(t+1) = \sqrt{V_{ri}(t)^2 + 2 \times \text{Acc}_{ri}(t) \times \text{Pop}_{ri}(t)}, \quad (4)$$

$$V_{le}(t+1) = \sqrt{V_{le}(t)^2 + 2 \times \text{Acc}_{le}(t) \times \text{Pop}_{le}(t)}, \quad (5)$$

where $V_f(t+1)$, $V_{ri}(t+1)$, and $V_{le}(t+1)$ denote velocity at a time $(t+1)$ for lead, right, and left dogs. $V_f(t)$, $V_{ri}(t)$, and $V_{le}(t)$ denote velocity at a time (t) for lead, right, and left dogs. $\text{Acc}_f(t)$, $\text{Acc}_{ri}(t)$, and $\text{Acc}_{le}(t)$ denote acceleration of the lead, right and left dogs. $\text{Pop}_f(t)$, $\text{Pop}_{ri}(t)$, and $\text{Pop}_{le}(t)$ denote positions of the lead, right, and left dogs.

Step 3b: The velocity of the sheep is calculated using the three herding techniques namely gathering, stalking, and eyeing based on the value of D_g given in the following equation:

$$D_g = (\text{fit}_f - \text{fit}_s) - ((\text{fit}_{le} + \text{fit}_{ri}) - \text{fit}_s), \quad (6)$$

where D_g compares the fitness of sheep to that of the fitness of lead dog and mean fitness of left and right dogs, fit_f is the fitness of lead dog at time (t), fit_{le} is the fitness of left dog at time (t), fit_{ri} is the fitness of right dog at time (t), and fit_s is the fitness of sheep at time (t).

(i) If the value of D_g is positive, it indicates that the sheep is nearer to the lead dog. In this case, the velocity of the gathering sheep is calculated using the following equation:

$$V_{sg}(t+1) = \sqrt{V_f(t+1)^2 + 2 \times \text{Acc}_f(t) \times \text{Pop}_{sg}(t)}, \quad (7)$$

where $V_{sg}(t+1)$ is the velocity of the gathered sheep at a time $(t+1)$, $V_f(t+1)$ is the velocity of the lead dog at a time $(t+1)$, $\text{Acc}_f(t)$ is the acceleration of the lead dog at a time (t), and $\text{Pop}_{sg}(t)$ is the location of the gathered sheep at a time (t).

(ii) If the value of D_g is negative, it indicates that the sheep is nearer to the left and right dogs. In this case, the velocity of the stalking sheep is calculated using the equations (8) to (10):

$$V_{ri} = \sqrt{\left(V_{ri}(t+1)\tan\ (\theta_1)\right)^2 + 2 \times Acc_{ri}(t) \times Pop_{ri}(t)} \quad (8)$$

where $V_{ri}(t+1)$ is velocity of the right dog at time $(t+1)$, $\mathrm{Acc}_{ri}(t)$ is the acceleration of the right dog at time (t), $\mathrm{Pop}_{ri}(t)$ is the location of the right dog at time (t), and θ_1 is the random angle between right dog and stalked sheep.

$$V_{le} = \sqrt{\left(V_{le}(t+1)\tan\ (\theta_2)\right)^2 + 2 \times Acc_{le}(t) \times Pop_{le}(t)} \quad (9)$$

where $V_{le}(t+1)$ is velocity of the left dog at time $(t+1)$, $\mathrm{Acc}_{le}(t)$ is the acceleration of the left dog at time (t), $\mathrm{Pop}_{le}(t)$ is the location of the left dog at time (t), and θ_2 is the random angle between left dog and stalked sheep.

$$V_{ss}(t+1) = \frac{V_{le} + V_{ri}}{2}, \quad (10)$$

where $V_{ss}(t+1)$ is the velocity of staked sheep at time $(t+1)$, V_{le} is the velocity of the left dog, and V_{ri} is the velocity of the right dog.

(iii) The sheep that have entirely lost their way must be kept an eye on. The velocity calculation of the eyeing sheep is computed using the following equation:

$$V_{se}(t+1) = \sqrt{V_{le}(t+1)^2 - 2 \times \mathrm{Acc}_{le}(t) \times \mathrm{Pop}_{le}(t)}, \quad (11)$$

where $V_{se}(t+1)$ is the velocity of the eyed sheep at time $(t+1)$, $V_{le}(t+1)$ is velocity of the left dog at time $(t+1)$, $\mathrm{Acc}_{le}(t)$ is the acceleration of the left dog at time (t), and $\mathrm{Pop}_{le}(t)$ is the location of the left dog at time (t).

Step 3c: The acceleration calculation of dogs and sheep is given in the following equation:

$$\mathrm{Acc}_i(t+1) = \frac{\left(V_i(t+1) - V_i(t)\right)}{\mathrm{Time}_i(t)}, \quad (12)$$

where $\mathrm{Acc}_i(t+1)$ is the acceleration of all dogs and sheep, viz., $i_f(t+1)$, $\mathrm{Acc}_{le}(t+1)$, $\mathrm{Acc}_{ri}(t+1)$, $\mathrm{Acc}_{sg}(t+1)$, $\mathrm{Acc}_{ss}(t+1)$ and $\mathrm{Acc}_{se}(t+1)$, and $\mathrm{i} \in \{f, le, ri, sg, ss\, to\, se\}$

Step 3d: The time calculation of dogs and sheep is given in the following equation:

$$\mathrm{Time}_i(t+1) = \mathrm{Avg}\sum_{i=1}^{d} \frac{\left(V_i(t+1) - V_i(t)\right)}{\mathrm{Acc}_i(t+1)}, \quad (13)$$

where $\mathrm{Time}_i(t+1)$ is the average time of traversal of each individual of dimension d.

Step 4: Update the population of dogs using equation (14) to (16):

$$\mathrm{Pop}_f(t+1) = V_f(t+1) \times \mathrm{Time}_f(t+1) + \frac{1}{2}\mathrm{Acc}_f(t+1) \times \mathrm{Time}_f(t+1)^2, \quad (14)$$

where $\mathrm{Pop}_f(t+1)$ is the location of lead dog at time $(t+1)$, $V_f(t+1)$ is the velocity of the lead dog at time $(t+1)$, $\mathrm{Time}_f(t+1)$ is the time required by the lead dog to move to $\mathrm{Pop}_f(t+1)$, and $\mathrm{Acc}_f(t+1)$ is the acceleration of lead dog at time $(t+1)$.

$$\mathrm{Pop}_{le}(t+1) = V_{le}(t+1) \times \mathrm{Time}_{le}(t+1) + \frac{1}{2}\mathrm{Acc}_{le}(t+1) \times \mathrm{Time}_{le}(t+1)^2, \quad (15)$$

where $\mathrm{Pop}_{le}(t+1)$ is the location of left dog at time $(t+1)$, $V_{le}(t+1)$ is the velocity of the left dog at time $(t+1)$, $\mathrm{Time}_{le}(t+1)$ is the time required by the left dog to move to $\mathrm{Pop}_f(t+1)$, and $\mathrm{Acc}_{le}(t+1)$ is the acceleration of left dog at time $(t+1)$.

$$\mathrm{Pop}_{ri}(t+1) = V_{ri}(t+1) \times \mathrm{Time}_{ri}(t+1) + \frac{1}{2}\mathrm{Acc}_{ri}(t+1) \times \mathrm{Time}_{ri}(t+1)^2, \quad (16)$$

where $\mathrm{Pop}_{ri}(t+1)$ is the location of right dog at time $(t+1)$, $V_{ri}(t+1)$ is the velocity of the right dog

at time $(t+1)$, $\text{Time}_{ri}(t+1)$ is the time required by the right dog to move to $\text{Pop}_f(t+1)$, and $\text{Acc}_{ri}(t+1)$ is the acceleration of right dog at time $(t+1)$.

Step 5: Update the population of sheep using equation (17) to (19):

$$\text{Pop}_{sg}(t+1) = V_{sg}(t+1) \times \text{Time}_{sg}(t+1) + \frac{1}{2}\text{Acc}_{sg}(t+1) \times \text{Time}_{sg}(t+1)^2, \tag{17}$$

where $\text{Pop}_{sg}(t+1)$ is the location of gathered sheep at time $(t+1)$, $V_{sg}(t+1)$ is the velocity of gathered sheep at time $(t+1)$, $\text{Time}_{sg}(t+1)$ is the time required by the gathered sheep to move to $\text{Pop}_f(t+1)$, and $\text{Acc}_{sg}(t+1)$ is the acceleration of gathered sheep at time $(t+1)$.

$$\text{Pop}_{ss}(t+1) = V_{ss}(t+1) \times \text{Time}_{ss}(t+1) - \frac{1}{2}\text{Acc}_{ss}(t+1) \times \text{Time}_{ss}(t+1)^2, \tag{18}$$

where $\text{Pop}_{ss}(t+1)$ is the location of stalked sheep at time $(t+1)$, $V_{ss}(t+1)$ is the velocity of stalked sheep at time $(t+1)$, $\text{Time}_{ss}(t+1)$ is the time required by the stalked sheep to move to $\text{Pop}_f(t+1)$, and $\text{Acc}_{ss}(t+1)$ is the acceleration of stalked sheep at time $(t+1)$.

$$\text{Pop}_{se}(t+1) = V_{se}(t+1) \times \text{Time}_{se}(t+1) - \frac{1}{2}\text{Acc}_{se}(t+1) \times \text{Time}_{se}(t+1)^2, \tag{19}$$

where $\text{Pop}_{se}(t+1)$ is the location of eyed sheep at time $(t+1)$, $V_{se}(t+1)$ is the velocity of eyed sheep at time $(t+1)$, $\text{Time}_{se}(t+1)$ is the time required by the eyed sheep to move to $\text{Pop}_f(t+1)$, and $\text{Acc}_{se}(t+1)$ is the acceleration of eyed sheep at time $(t+1)$.

Step 6: Repeat steps 2 to 5 until the solution converges or maximum number of iterations is reached. The rule subset with maximum classification accuracy obtained by the k-NN classifier is treated as the optimal subset.

Output: optimal rule subset

The rule selection uses Border Collie bio-inspired algorithm to determine optimal rule subsets. BCO starts with a random set of solutions. Each rule or solution is evaluated using the accuracy of the k-NN classifier. The k-NN finds the k-nearest neighbours to perform rule selection. The parameters of BCO are tuned after repeated experiments. The algorithm is set for different runs and a different number of rules are selected at an individual run. The values are varied to analyse how the parameters affect the algorithm's performance. The process is repeated until the solution converges, or 100 iterations are performed to determine the best set of rules. After selecting the best rules, they are stored in Rule Base. Table 13 represents the number of rules before and after performing rule selection. Table 14 represents a subset of rules from the optimal rule subset.

4. Results and Discussion

The work has been implemented in Java 1.8. The experiments are conducted using open-source software, namely jEdit, Nutch, Lucene, and Rhino. jEdit is a text editor with hundreds of developing plugins and functionality that are reliable and easy to use. Nutch is an open-source Java search engine implementation. Lucene is software for information retrieval. Rhino is a JavaScript interpreter and compiler for the Mozilla/Firefox browser written in Java. Table 15 represents the number of classes, code smells, and lines of code for the software used.

The proposed work selects the optimal rule subsets using Border Collie bio-inspired algorithm with the accuracy of the k-NN classifier as the fitness function. The algorithm selects rules depending on the performance of the classifier. The rules are selected based on the accuracy of the k-NN classifier, which is largely dependent on the k-value. In order to determine the optimum value of k, a range of k-values between 1 and 30 is used, and the error mean rate is calculated for each k. Since the error rate does not vary significantly after $k = 17$ as shown in Figure 3, $k = 17$ is chosen as the threshold value of k. At optimal k-value, the model accuracy is improved.

The proposed algorithm uses BCO that selects 35 rules out of 58 in jEdit, 12 rules out of 19 in Nutch, 9 rules out of 15 in Lucene, and 16 rules out of 27 rules in Rhino, respectively. The performance of the proposed work is measured in terms of accuracy, precision, recall, specificity, and defined using equations (20)–(23):

Table 13: Outline of the number of rules.

Software	No. of rules	
	Before rule selection	After rule selection (BCO)
jEdit	58	35
Nutch	19	12
Lucene	15	9
Rhino	27	16

Table 14: Subset of rules from optimal rule subset.

Rules for detection	Code smell
IF ObjectCreator AND FieldReader	BL
IF Exception AND ArrayReader	FE
IF FieldWriter AND LocalWriter	FD
IF ArrayWriter AND Looping AND LocalWriter	SC
IF ArrayReader AND ArrayWriter	DC

BL: blob, FE-feature envy, FD: functional decomposition, SC: spaghetti code, and DC: data class.

Table 15: Outline of software.

Software	Release	Number of classes	Number of smells	KLOC
jEdit	v.5.1.0	316	27	101
Nutch	v.1.1	207	79	39
Lucene	v.1.4.3	154	41	33
Rhino	v.1.7R1	305	82	57

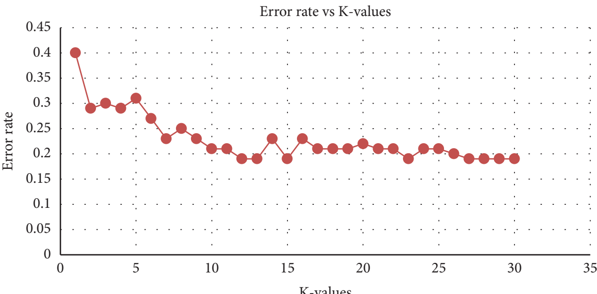


Figure 3: Error rate for range of k-values.

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{FP} + \text{TN} + \text{FN}}, \quad (20)$$

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}, \quad (21)$$

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}, \quad (22)$$

$$\text{Specificity} = \frac{\text{TN}}{\text{TN} + \text{FP}}, \quad (23)$$

where TP is the number of true positives, TN denotes the number of accurately predicted negatives, FP denotes the number of negatives predicted as positives, and FN denotes the number of positives predicted as negatives. The confusion matrix obtained for jEdit, Nutch, Lucene, and Rhino are shown in Table 16. The competitive results achieved from the aforementioned software are shown in Table 17. The proposed work produced an accuracy of 98.78% for jEdit, 97.45% for Nutch, 95.58% for Lucene, and 96.34% for Rhino, respectively. The proposed work is compared to other well-known techniques for code smell detection (see Table 18), as the authors used the same open-source software to detect code smells. However, the techniques PSO, GP, GA, PEA, and HPSOM express rules as a set of software metrics and thresholds rather than set of nanopatterns. Software metrics are measures of software characteristics. Nanopatterns are traceable fundamental characteristics of a method or a procedure. When considering rules from both the

Table 16: Confusion matrix for jEdit, Nutch, Lucene, and Rhino.

Software	Actual and predicted	Correctly identified	Incorrectly identified
jEdit	Correctly identified	23	4
	Incorrectly identified	4	76
Nutch	Correctly identified	72	6
	Incorrectly identified	6	309
Lucene	Correctly identified	36	3
	Incorrectly identified	62	159
Rhino	Correctly identified	76	7
	Incorrectly identified	5	322

Table 17: Performance measures.

Software	Accuracy (%)	Recall (%)	Specificity (%)	Precision (%)
jEdit	98.78	95.34	96.39	93.71
Nutch	97.45	94.87	98.09	92.50
Lucene	95.58	92.30	96.36	85.71
Rhino	96.34	92.85	97.05	86.66

Table 18: Comparison of proposed work with existing works.

Software	PSO		GP		GA		PEA		HPSOM		Proposed work	
	PR	RE	PR	RE	PR	RE	PR	RE	PR	RE	PR	RE
jEdit	86	90	84	77	77	85	87	88	92	**95**	**96**	**95**
Nutch	90	86	82	86	83	84	88	87	90	92	**93**	**95**
Lucene	92	82	84	78	82	84	91	86	**94**	88	86	**92**
Rhino	89	93	81	84	86	81	89	92	**95**	**95**	87	93

PR: precision and RE: recall. Bold values denote the highest precision and recall values.

characteristics, competitive recall, and precision values are seen for nanopatterns, as shown in Table 18.

5. Conclusion and Scope for Future

The proposed work to detect code smells using rules extracted from nanopatterns has been designed and implemented. Nanopatterns and the class labels are extracted from the software. Rules are generated from the input nanopatterns using the Apriori algorithm and associated with the class labels. Rule selection has been performed using the BCO algorithm with the k-NN classifier. The accuracy of the k-NN classifier was employed as the fitness function to maximize the classification accuracy of the rule set. The optimal rule subsets that are selected by the BCO algorithm are stored in the Rule Base. The framework has been trained and tested using software, namely jEdit, Nutch, Lucene, and Rhino. Compared to other works, the proposed work produced a precision of 90% and a recall of 93%.

Future research would incorporate additional code smells recommended by Fowler to ensure its applicability. The research would also incorporate design patterns and micropatterns to detect code smells. Although this work only focuses on detecting code smells, it can be enhanced to include corrective approaches as well.

Data Availability

The data for this study are available from previously reported studies. The open-source software used in this research is available at Qualitas Corpus https://qualitascorpus.com/. The authors confirm that the codes used in this study are available on GitHub at https://github.com/IRTJULIET/Nano-Patterns-Code-Smells/commits/V1.1.

Conflicts of Interest

The authors declare that they have no conflicts of interest regarding the publication of this paper.

Acknowledgments

The authors thank Visvesvaraya Ph.D. Scheme for Electronics and IT for the financial support of the research work.

References

[1] A. Abran and H. Nguyenkim, "Measurement of the maintenance process from a demand-based perspective," *Journal of Software Maintenance: Research and Practice*, vol. 5, no. 2, pp. 63–90, 1993.

[2] M. Fowler, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley Longman, Westford, MA, USA, 1999.

[3] Z. Soh, A. Yamashita, F. Khomhand, and Y. G. Guéhéneuc, "Do code smells impact the effort of different maintenance programming activities?"vol. 1, pp. 393–402, in *Proceedings of the 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1, pp. 393–402, IEEE, Osaka, Japan, May 2016.

[4] J. P dos Reis, F. B. e Abreu, G. de Figueiredo Carneiroand, and C. Anslow, "Code smells detection and visualization: a systematic literature review," *Archives of Computational Methods in Engineering*, vol. 29, pp. 1–48, 2021.

[5] F. Khomh, S. Vaucher, Y. G. Guéhéneuc, and H. Sahraoui, "A bayesian approach for the detection of code and design smells," in *Proceedings of the 2009 Ninth International Conference on Quality Software*, pp. 305–314, IEEE, Jeju, Korea (South), January 2009.

[6] F. Arcelli Fontana, M. V. Mäntylä, M. Zanoni, and A. Marino, "Comparing and experimenting machine learning techniques for code smell detection," *Empirical Software Engineering*, vol. 21, no. 3, pp. 1143–1191, 2016.

[7] D. J. Thessalonica, H. K. Nehemiah, S. Sreejith, and A. Kannan, "Metric-based rule optimizing system for code smell detection using salp swarm and cockroach swarm algorithm," *Journal of Intelligent & Fuzzy Systems*, vol. 43, no. 6, pp. 7243–7260, 2022.

[8] U. Mansoor, M. Kessentini, M. Wimmer, and K. Deb, "Multiview refactoring of class and activity diagrams using a multiobjective evolutionary algorithm," *Software Quality Journal*, vol. 25, no. 2, pp. 473–501, 2017.

[9] A. Trivedi, J. S. Thakur, and A. Gupta, "Code nano-pattern detection using deep learning," in *Proceedings of the 13th Innovations in Software Engineering Conference on Formerly known as India Software Engineering Conference*, pp. 1–6, NewYork, NY, USA, February 2020.

[10] D. Yu, J. Yang, X. Chen, and J. Chen, "Detecting java code clones based on bytecode sequence alignment," *IEEE Access*, vol. 7, pp. 22421–22433, 2019.

[11] J. Gill and I. Maman, "Micro patterns in Java code," in *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pp. 97–116, San Diego, CA, USA, October 2005.

[12] E. W. Host and B. M. Ostvold, "The programmer's lexicon, volume I: the verbs," in *Proceedings of the Seventh IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2007)*, pp. 193–202, IEEE, Washington, D.C, USA, September 2007.

[13] J. Singer, G. Brown, M. Luján, A. Pocock, and P. Yiapanis, "Fundamental nano patterns to characterize and classify java methods," *Electronic Notes in Theoretical Computer Science*, vol. 253, no. 7, pp. 191–204, 2010.

[14] E. Bruneton, R. Lenglet, and T. Coupaye, "ASM: a code manipulation tool to implement adaptable systems," *Adaptable and extensible component systems*, vol. 30, no. 19, 2002.

[15] K. Z. Sultana, A. Deo, and B. J. Williams, "Correlation analysis among java nano patterns and software vulnerabilities," in *Proceedings of the 2017 IEEE 18th International Symposium on High Assurance Systems Engineering (HASE)*, pp. 69–76, IEEE, Singapore, April 2017.

[16] K. Z. Sultana, B. J. Williams, and A. Bosu, "A comparison of nano patterns Vs. Software metrics in vulnerability prediction," in *Proceedings of the 2018 25th Asia-Pacific Software Engineering Conference (APSEC)*, pp. 355–364, IEEE, Nara, Japan, May 2018.

[17] M. Dahm, *Byte Code Engineering*Springer, Berlin, Heidelberg, 1999.

[18] A. White, "Serp," 2007, http://Http://Serp.Sourceforge.Net.

[19] G. A. Cohen and D Kaminsky, "Automatic program transformation WithJOIE," in *Proceedings of the USENIX Annual Technical Conference*, vol. 98, NewYork, NY, USA, June 1998.

[20] J. Z. C. Lu and B. Xu, "A toolkit for Java bytecode analysis," in *Proceedings of the Seventh IASTED International Conference on Software Engineering and Applications*, pp. 482–487, Berlin, Heidelberg, January 2003.

[21] A. D. S. Mignon and R. L. de Azevedo da Rocha, "An application of composite nano patterns to compiler selected profiling techniques," in *Proceedings of the 6th International Conference on Software and Computer Applications*, pp. 186–190, NewYork, NY, USA, February 2017.

[22] F. Batarseh, "Java nano patterns: a set of reusable objects," in *Proceedings of the 48th Annual Southeast Regional Conference*, pp. 1–4, Oxford, MS, USA, April 2010.

[23] G. Saranya, H. K. Nehemiah, and A. Kannan, "Hybrid particle swarm optimisation with mutation for code smell detection," *International Journal of Bio-Inspired Computation*, vol. 12, no. 3, pp. 186–195, 2018.

[24] G. Saranya, H. Khanna Nehemiah, A. Kannan, and V. Nithya, "Model level code smell detection using egapso based on similarity measures," *Alexandria Engineering Journal*, vol. 57, no. 3, pp. 1631–1642, 2018.

[25] C. Tjortjis, "Mining Association Rules from Code (MARC) to support legacy software management," *Software Quality Journal*, vol. 28, no. 2, pp. 633–662, 2019.

[26] K. D. Rajab, "New associative classification method based on rule pruning for classification of datasets," *IEEE Access*, vol. 7, pp. 157783–157795, 2019.

[27] L. Zhang, "Associative classification using an immune optimization algorithm," in *Proceedings of the 2012 IEEE International Conference on Automation and Logistics*, pp. 179–184, IEEE, Zhengzhou, China, Augest 2012.

[28] J. Mattiev and B. Kavsek, "Coverage-based classification using association rule mining," *Applied Sciences*, vol. 10, no. 20, p. 7013, 2020.

[29] H. H. Awan and W. Shahzad, "Semi-supervised associative classification using ant colony optimization algorithm," *Peer Journal Computer Science*, vol. 7, p. 676, 2021.

[30] N. Pritam, M. Khari, R. Kumar et al., "Assessment of code smell for predicting class change proneness using machine learning," *IEEE Access*, vol. 7, pp. 37414–37425, 2019.

[31] M. Khari, P. Kumar, D. Burgos, and R. G. Crespo, "Optimized test suites for automated testing using different optimization techniques," *Soft Computing*, vol. 22, no. 24, pp. 8341–8352, 2018.

[32] M. Khari, A. Sinha, E. Herrerra-Viedma, and R. G. Crespo, "On the use of meta-heuristic algorithms for automated test suite generation in software testing," in *Toward Humanoid Robots: The Role of Fuzzy Sets*, pp. 149–197, Springer, Cham, Champa, 2021.

[33] L. H. Son, N. Pritam, M. Khari, R. Kumar, P. M. Phuong, and P. H. Thong, "Empirical study of software defect prediction: a systematic mapping," *Symmetry*, vol. 11, no. 2, p. 212, 2019.

[34] Y. Zhang, C. Ge, S. Hong, R. Tian, C. Dong, and J. Liu, "DeleSmell: code smell detection based on deep learning and latent semantic analysis," *Knowledge-Based Systems*, vol. 255, Article ID 109737, 2022.

[35] S. Boutaib, M. Elarbi, S. Bechikh, C. A. C Coello, and L. B. Said, "Uncertainty-wise software anti-patterns detection: a possibilistic evolutionary machine learning approach," *Applied Soft Computing*, vol. 129, Article ID 109620, 2022.

[36] R. Agrawal, T. Imieliński, and A. Swami, "Mining association rules between sets of items in large databases," in *Proceedings of the 1993 ACM SIGMOD international conference on Management of data*, pp. 207–216, Washington, D.C, USA, January 1993.

[37] C. Aflori and M. Craus, "Grid implementation of the Apriori algorithm," *Advances In Engineering Software*, vol. 38, no. 5, pp. 295–300, 2007.

[38] T. Dutta, S. Bhattacharyya, S. Dey, and J. Platos, "Border collie optimization," *IEEE Access*, vol. 8, pp. 109177–109197, 2020.