Unsupervised Machine Learning for Effective Code Smell Detection: A Novel Method

Ruchin Gupta, Narendra Kumar, Sunil Kumar, and Jitendra Kumar Seth

Original scientific article

***Abstract*—The quality of source code is negatively impacted by code smells. Since the term "code smell" originated, numerous attempts have been made to comprehend it by identifying it using various techniques, such as metric-based, heuristic-based, optimization-based, machine learning (ML)-based, etc. Among these, supervised machine learning (SML) has shown effectiveness in detecting code smells. However, SML techniques have significant limitations, including the dependency on expensive and high-quality labeled data, the need for representative training datasets, and the risk of introducing biases in labeled examples that lead to skewed predictions. To overcome these challenges, this study introduces a method that leverages unsupervised machine learning (UnML) along with feature engineering. Unlike SML, UnML does not require labeled data and minimizes potential biases. The proposed method was evaluated using four datasets containing different types of code smells and was compared with a previous study that used SML techniques. The results indicate that the UnML-based method is effective, achieving outcomes closely aligned with those from the SML approach. This method is especially beneficial in situations where labeled data is scarce or unavailable and can be used to identify new code smells, generate labeled data for SML and detect multiple code smells simultaneously within a codebase.**

***Index terms*—code smell, unsupervised machine learning, open-source Java projects.**

I. Introduction

Kent Beck was the first person who invented the phrase “code smell” in 1999 [1]. Code smell was defined as “certain structures in the code that suggest (or sometimes scream) for refactoring.” Code smell refers to any characteristic of a source code that could indicate a significant underlying issue. The term "smell" is used metaphorically to indicate that there might be something wrong with the code, even though it may still function correctly. Code smells are not bugs themselves, but they often indicate areas of the code that could benefit from refactoring or further investigation to improve maintainability, readability, or performance.

Manuscript received October 18, 2024; revised November 6, 2024. Date of publication December 9, 2024. Date of current version December 9, 2024. The associate editor prof. Renata Lopes Rosa has been coordinating the review of this manuscript and approved it for publication.

R. Gupta and J. K. Seth are with the Department of Information Technology, KIET Group of Institutions, Delhi-NCR, Ghaziabad, India.(e-mails: skg11in@yahoo.co.in, drjkseth@gmail.com).

N. Kumar and S. Kumar are with the Galgotias College of Engineering and Technology (e-mails: nkteotia2004@gmail.com, skkiet@gmail.com).

Digital Object Identifier (DOI): 10.24138/jcomss-2024-0083

Code smells indicate symptoms present in the source code that have an impact on the quality characteristics of the software [1], [2]. Code smells are apparent characteristics that suggest the existence of design problems or deficiencies in the code. These problems or deficiencies have a significant impact on crucial aspects of code quality, like maintainability, reusability, and understandability [1]. Code smell typically refers to underlying problems in a code that degrade its quality and can lead to significant issues [3]–[5]. Code smells when present in large numbers in software makes it difficult to maintain. First, Martin Fowler defined a catalogue of 22 code smells [1].

The first step in dealing with code smell is its detection which is followed by refactoring. The process of refactoring modifies the internal structure of code without altering its external functionality. The main objective of refactoring is enhancing the code’s structure to facilitate its comprehension, modification, and maintenance while retaining its functionality.

Code smell detection is crucial for several reasons. Identifying and addressing code smells helps improve the maintainability of software systems. By detecting and refactoring these smells, developers can ensure that the codebase remains clean, understandable, and easier to maintain over time. Code smells can be indicators of potential bugs or defects in the code. Addressing code smells at an early stage in the development process helps to prevent the accumulation of technical debt and reduce the likelihood of introducing bugs or errors. This ultimately leads to higher-quality software products that are more reliable and less prone to unexpected issues. Code smells such as duplicated code, long methods, or excessive coupling can negatively impact the scalability and performance of software systems. Clean, well-structured code is more reusable than code with numerous smells. Working with a codebase riddled with code smells can be frustrating and demoralizing for developers. Addressing code smells and maintaining a clean, well-structured codebase can boost developer morale, leading to a more positive and productive work environment.

Since the inception of the concept of code smell, the existing body of literature demonstrates that numerous attempts have been undertaken to detect them through diverse approaches. Multiple approaches have been employed to identify and detect different types of code smells. The literature on code smell detection has identified five categories of approaches: “metrics-based, rules or heuristic-based, code change information-based, machine learning-based, and

optimization algorithm-based"[6]. The literature review conducted by Azeem et al. [7] identified a total of 13 code smells (out of 22 code smells from Martin Fowler's catalog) that have been targeted for code smell detection using machine learning (ML) techniques. Based on a survey conducted by Al-Shaaby [8], the existing literature indicates that the methods employed in the domain of code smell for their detection have been based on SML techniques. Furthermore, based on the information available to us, there is no study conducted on the utilization of UnML for code smell detection. SML has several limitations. SML relies on labelled data which can be expensive and time-consuming to obtain. Supervised learning learns patterns and makes predictions based on input features. SML focuses on minimizing prediction errors by learning from labeled examples, which may not capture all the nuances of the underlying data distribution. In SML, performance relies on the quality and representativeness of labeled training data, which may not fully encompass the intricacy of real-world situations. Biases in the labeled examples can lead to biased model predictions.

The use of the unsupervised machine learning (UnML) technique can offer several advantages over the SML technique in for identification of code smells. UnML methods don't require labeled examples, enabling them to analyze large codebases without the need for manual labeling. They can uncover hidden structures and patterns within codebases, identifying not only well-known smells but also novel or subtle issues that might be missed by manual inspection or predefined labels. This capability allows for a more thorough comprehension of the overall code quality. UnML approaches provide a more objective assessment of code quality by learning directly from the data distribution. This reduces the risk of bias introduced by labeled examples and allows for a more data-driven understanding of code smells. UnML scales more effectively to large codebases since they don't require manual labeling. UnML methods are more flexible and adaptable, capable of detecting various types of code smells without explicit guidance or predefined labels. This adaptability makes them suitable for detecting both known and unknown smells, as well as for exploring new types of code quality issues.

This study represents the utilization of an UnML to detect code smells. Based on our current knowledge, it is the first study of its kind to explore the feasibility of an UnML algorithm for the identification of code smell. Thus, this paper outlines the following contributions:

- This study proposes a method that uses a self-organizing map (SOM), an UnML algorithm, along with feature engineering to identify code smells.
- The proposed method is evaluated on 4 popular and publicly available datasets [9] of 4 different code smells: Long method, feature envy, god class, and data class. The proposed method's performance has been assessed using commonly used performance measures: precision, recall, F-measure, accuracy, MCC, and AUC-ROC. The research findings along with results have been made accessible to the research community for future investigation.
- The proposed method's results employing an UnML algorithm have been contrasted with the results previous study [9] which has used several (16) SML algorithms. The study conducted by Fontana [9] is a crucial and notable advancement in the domain of code smell through the application of SML algorithms.

The subsequent text outlines the structure of the article. The second section of the paper outlines the related work for code smell that uses machine learning for their detection. Section III elaborates on the proposed method mentioned in this study. Section IV provides the details about the experimental datasets employed in the study. Section V provides a detail of the UnML algorithm used. The performance measures used to determine the effectiveness of the proposed method are elaborated in Section VI. The results of the conducted experiments are analyzed in Section VII. Section VIII provides the last thoughts and delineates opportunities for future research.

II. Related Work

This section summarizes the observed significant literature on code smell detection using the ML technique.

Kreimer [10] employed a decision tree-based adaptive approach to identify long methods and large class code smells, integrating object-oriented metrics with machine learning to autonomously detect design flaws. This supervised learning technique utilized a program dependency graph (PDG) as an abstract representation of the program, where specific metric values were fed into the trained model to identify design flaws. Metrics such as statement count, method complexity, parameter count, and local variable count were used to detect long methods.

Authors [11] utilized a Bayesian approach, a probabilistic model, to identify code smells by computing the likelihood of a class being associated with a particular smell. This method also falls under the category of supervised learning, as it relies on known probabilities to predict code smells.

Sérgio et al. [12] applied binary logistic regression, a supervised learning technique, to identify long methods. They used metrics like Method Lines of Code and cyclomatic complexity as regressors. The model required an initial training phase with expert-labeled data to calibrate the classification process.

Khomh et al. [13] introduced BDTEX, a Bayesian Decision-Theoretic model using Bayesian Networks (BBNs), to identify antipatterns in software. This supervised learning method and modeled symptoms at the operational level to predict the likelihood of a class being an antipattern, demonstrating strong performance on well-structured programs.

Abdou et al. [14] employed a Support Vector Machine (SVM), a supervised learning technique, trained on 60 object-oriented metrics to detect design smells like blob, feature concentration, and spaghetti code. Their novel method, SVMDetect, trained the SVM model on a dataset of object-

oriented metrics and applied it to detect design smells in software classes.

Fontana et al. [9] used 16 different machine learning algorithms, all under the umbrella of supervised learning, to detect code smells such as data class, god class, feature envy, and long method across 74 systems. Metrics were computed using the DFMJ tool, and the performance of these algorithms was evaluated on the dataset.

Kim [15] utilized a neural network implemented in TensorFlow, a supervised deep learning technique, to predict seven different code smells based on the analysis of object-oriented metrics from 20 open-source Java projects. The neural network was trained on these metrics, with calculations performed across various epochs and hidden layers to enhance detection accuracy.

In contrast, a study [16] introduced a hybrid detection approach using a deep autoencoder and Artificial Neural Network (ANN). The deep autoencoder, an unsupervised learning method, performed dimensionality reduction, which was then followed by supervised learning through the ANN to detect code smells like God Class and Feature Envy. This approach demonstrated improved accuracy by combining unsupervised and supervised learning.

Another study [17] proposed a deep learning-based approach to detect feature envy code smells using Convolutional Neural Networks (CNNs), a supervised learning technique. The study also developed an automatic method for generating labeled training data, utilizing both structural and textual information for training the neural network classifier.

The study [18] introduced SMAD, an ensemble method combining multiple supervised machine learning classifiers, including a Multi-layer Perceptron, to detect anti-patterns like God Class and Feature Envy. The method aggregated core metrics from different detection tools as input for the machine learning model, which demonstrated superior performance in comparison to other ensemble methods.

Gupta et al. [19] investigated the use of six different supervised machine learning algorithms, including Naive Bayes, KNN, MLP, Decision Tree, Logistic Regression, and Random Forest, to detect code smells in four datasets. The study highlighted the impact of feature selection and parameter optimization on model accuracy, particularly for algorithms like Random Forest and Logistic Regression.

Vatanapakorn et al. [20] employed eight supervised learning algorithms to detect code smells in Python programs, enhancing performance through correlation-based feature selection and forward stepwise selection methods, which helped in identifying the most relevant software metrics for each code smell category.

Two studies [21], [22] explored the use of transfer learning, a domain adaptation technique, for code smell detection using deep learning models like 1D and 2D CNNs, RNNs, and Autoencoders. These models utilized tokenized source code sequences as input, demonstrating the potential of both supervised learning.

Recently, Gupta [23] demonstrated the potential of a customized transfer learning method, "MDITKL," for detecting code smells in heterogeneous data. This method is a variant of domain invariant transfer kernel learning[24], a homogeneous transfer learning technique, which adapts to the challenges of detecting long methods and temporary field code smells in diverse datasets. Another study by [25] identified two Python code smells—Large Class and Long Method—using five ML models. Similarly, [26] employed eight ML models, along with preprocessing techniques, to detect four code smells Blob, Long Method, Feature Envy, and Data Class. Thus, the subsequent observations were derived from the literature review.

- Java has been the most extensively studied language for detecting code smells utilizing machine learning methods.
- Research has been conducted on the application of ensemble methods. Ensemble methods were used to improve the performance of machine learning algorithms.
- Sophisticated machine learning methodologies such as deep learning and transfer learning, have exhibited their value in code smell detection and are poised to enhance their capabilities.
- No empirical study has employed an UnML approach to detect code smells.

Thus, the literature survey demonstrates that various SML methods have been employed for the identification of code smell. However, to date, there has been no study conducted utilizing UnML technique(s).

III. The Proposed Method

The following section provides a detailed discussion of the proposed experimental methodology. It has been kept simple as shown in Figure 1. It consists of 5 main steps.

Step 1 (download code smell datasets). Step 1 is collecting/downloading existing considered open-source Java project datasets under study from the web for experimentation.

Step 2 (Clean dataset(s)). Step 2 includes pre-processing the dataset through analysis and clean-up steps. The dataset has been subjected to a comprehensive examination to identify and address duplicate rows and columns, as well as missing values, to ensure data accuracy. The dataset has been processed to remove any duplicate rows and columns and to fill in missing values for completeness.

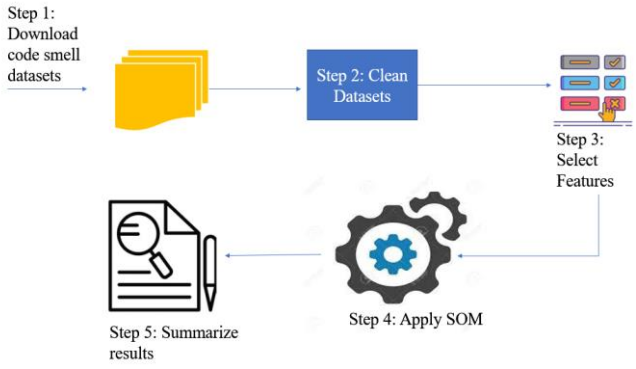

Fig. 1. The proposed method using UnML (SOM)

Step 3 (Select features). Step 3 selects features that are significant using a feature selection technique.

Features play a crucial role in the development of prediction models, including regression and classification models. Overfitting can occur in a model with a high feature count,

causing it to predict the training dataset only accurately. This is an undesired situation. By assigning a meaningful value to each feature, one can apply the feature importance technique to determine the most and least important features for prediction. Step 3 involves identifying the essential features when dealing with many features in the dataset(s). The significance of the chosen features was subsequently analyzed based on the defined code smell criteria. An open-source Python package called Featurewiz [27] was utilized in this study to identify crucial features due to its simplicity and efficiency in feature selection from a dataset.

Step 4 (Apply SOM). Step 4 applies SOM. The dataset is partitioned into training and testing datasets using an 80:20 ratio. The performance is analyzed using performance metrics like MCC, precision, recall, F-measure, accuracy, and AUC.

Step 5 (Summarize results). Step 5 involves the collection of experimental results and the preparation of a summary to conclude.

IV. Experimental Datasets

The Java programming language has experienced a surge in popularity starting from its inception, largely due to the widespread adoption of the Internet. Java is widely regarded as the most extensively studied programming language in terms of code smells, as evidenced by the literature [6], [28]. The study has used 4 extremely popular, manually validated, and published datasets over Java projects by Arcelli Fontana et al. [9] for 4 different code smells namely long method (LM), feature envy (FE), data class (DC), and god class (GC) because they were the only publicly accessible datasets with several metrics (computed over source code) and types of smells, they were judged appropriate for the experimentation in this paper.

The authors curated a dataset [9] consisting of 74 Java open-source software projects from varied application fields. These projects were picked from the Qualitas Corpus[29]. The various project metrics at the class and method level were computed using the tool DFMC4J. Subsequently, the authors applied smell detection tools namely “iPlasma, PMD1, Fluid Tool, AntiPattern Scanner” and rules to label code smells in projects followed through a manual check done by 3 trained students for the specified task. All these students first performed individual evaluations of code smells and then they discussed among themselves and came to a consensus. Their discussion produced a set of rules to determine each reported code smell. Therefore, a total of four datasets were created, each corresponding to LM, FE, DC, and GC. Appendix A shows the various features used at the class and method level. A comprehensive inventory of features and their corresponding definitions can be found in the appendix of the research paper authored by Arcelli Fontana [9].

The following steps were performed for the preparation of datasets for the experimentation.

1) The downloaded published datasets were assessed to gain insights into the various features employed in the datasets along with other properties such as their highest and lowest values, the count of total samples present in the datasets, and so forth. The study revealed that the datasets for LM and FE had the same features. Similarly, the datasets for GC and DC also had identical features. Table IV indicates that every dataset consisted of approximately 33% smelly samples (positive) and 67% non-smelly samples (negative).

2) The downloaded datasets were found to have several instances of missing values. Table I displays the number of missing values, the percentage of missing values, and the name of each feature together with the matching count of missing values for every downloaded dataset. The mean value approach is used to calculate missing data because of its simplicity and broad use as an imputation tool in research.

TABLE I
Code Smells Datasets Characteristics
(before feature selection)

SI No.	Dataset name	Number of features	No. of smelly samples	No. of –non-smelly samples	Count of missing values	Missing values %	Feature - count of missing values
1	DC	62	140	280	75	0.0028%	NMO-19, NIM -19, NOC -9, WOC -28
2	GC				76	0.0029%	NMO-20, NIM-20, NOC-8, WOC -28
3	LM	82			92	0.0026%	NOC-3, WOC-31 NIM-29, NMO-29
4	FE				92	0.0026%	NOC-3, WOC-31 NIM-29, NMO-29

TABLE II
Datasets characteristics after feature selection

Dataset name	Number of features	Name of features
DC_n	15	NOAM_type, ATFD_type, TCC_type, RFC_type, WMC_type, LCOM5_type, WOC_type, AMW_type, NOI_project, LOC_project, NOPA_type, number_public_visibility_methods, number_private_visibility_methods, number_final_methods, number_static_methods
GC_n	15	ATFD_type, RFC_type, LCOM5_type, WMCNAMM_type, AMW_type, LOC_package, NOPK_project, NOCS_project, num_static_attributes, number_private_visibility_methods, num_not_final_not_static_attributes, number_protected_visibility_methods, number_final_methods, number_static_methods, number_not_final_not_static_methods
LM_n	18	NOP_method, ATFD_method, CM_method, LOC_method, CYCLO_method, ATLD_method, CINT_method, CDISP_method, NOAM_type, NOA_type, LCOM5_type, WMCNAMM_type, AMW_type, NOCS_package, LOC_package, NOI_project, LOC_project, number_static_methods
FE_n	20	NOP_method, CC_method, ATFD_method, MAXNESTING_method, LOC_method, MaMCL_method, LAA_method, ATLD_method, CINT_method, NMO_type, ATFD_type, NOA_type, NOPA_type, CBO_type, NOI_project, num_final_not_static_attributes, isStatic_method number_public_visibility_methods, number_package_visibility_methods, number_static_methods

Following the application of step 3 (select features), which is covered in section III (the proposed method), table II shows the characteristics of the datasets. Table II shows the different features that the Featurewiz technique chose for each of the four code smells.

The shared features between GC and DC are displayed in blue, and between FE and LM in red in Appendix C. The features that have been mentioned for each of these four code smells—especially the common features—should be carefully considered by developers. The only feature that is chosen in all four code smells is number_static_methods. Therefore, when designing the code, due consideration should be paid to this feature.

V. UnML Algorithm Used

The literature shows that self-organizing maps have been employed in various applications such as texture classification [30], intrusion detection [31], pattern classification [32], image database classification [33], classification of dermatologic data [34], and water quality classification [35]. The study utilized a Self-Organizing Map for conducting the experimentation. A Self-Organizing Map (SOM), commonly referred to as a Kohonen map, is a form of artificial neural network classified under UnML algorithms. It was created by the Finnish researcher Teuvo Kohonen in the 1980s. A SOM is designed to project high-dimensional data onto a lower-dimensional space, typically 1D or 2D, while maintaining the topological relationships of the original data. Self-organizing maps do not rely on labeled training data for classification. Instead, they classify data based on spatial relationships within the input space. The steps of the SOM algorithm in the simplified form are outlined below:

Step1) (Initialization)

SOM is comprised of a grid of nodes organized in a one- or two-dimensional lattice. Each node is linked to a weight vector that has the same dimensionality as the input data. Weight vectors are commonly initialized through randomization or methods such as Principal Component Analysis.

Step 2) (Training)

During the training phase, a random sample is selected from the input data and fed into the Self-Organizing Map (SOM). A comparison is made between the weight vector of each node and the input vector. The node with the weight vector closest to the input vector is referred to as the "winning node" or "best-matching unit" (BMU). The BMU is calculated utilizing a distance metric, typically the Euclidean distance.

Step 3) (Neighbourhood Function)

The SOM utilizes a neighborhood function to determine the impact of training on neighboring nodes. Initially, the neighborhood function is usually initialized to cover the entire Self-Organizing Map (SOM) grid. The neighborhood function decreases over time as the SOM learns, usually following a decay schedule.

Step 4) (Weight Update)

Once the Best Matching Unit (BMU) is identified, the weights of both the BMU and its neighboring nodes are adjusted to align more closely with the input vector. The degree of adjustment is contingent on variables like the proximity to the BMU and the learning rate. Nodes in proximity to the Best Matching Unit (BMU) undergo more significant weight adjustments compared to more distant nodes. The learning rate diminishes gradually as training progresses, usually in accordance with a predefined decay schedule.

Step 5) (Iteration)

Steps 2-4 are iterated for multiple epochs, enabling the SOM to incrementally modify its weights and structure the input data in the lower-dimensional space. The number of iterations and the decay schedules for the learning rate and neighborhood function are dependent on factors such as data complexity and desired convergence rate.

During the training phase, the SOM algorithm learns to map the input data onto a lower-dimensional grid while maintaining the topological relationships of the original high-dimensional data. The process leads to the creation of a map in which similar input data points are positioned near each other, making Self-Organizing Maps (SOMs) suitable for clustering and classification.

Step 6) (Classification)

After training, the SOM is ready to be used for classification. For each input data point, BMU is computed to determine the closest neuron in the SOM grid. Input data point is assigned to the cluster represented by the BMU's location in the grid.

For the experimentation, the study has used a Python package called MiniSom. MiniSom (Mini SOM) [36] is a variation of the SOM algorithm, also known as Kohonen maps. MiniSom typically offers a more memory-efficient and computationally lighter implementation compared to traditional SOMs. In MiniSom, like in traditional Self-Organizing Maps (SOMs), several parameters are adjusted to control the behavior of the algorithm and the resulting map. Table III displays the values of different parameters used during implementation. These parameters provide control over the training process and the properties of the resulting SOM. Here are some of the most common parameters (given in Table III) used in MiniSom:

a) Grid Size (m, n):

MiniSom creates a grid of neurons with dimensions m x n. This grid represents the layout of the SOM, where each neuron corresponds to a specific location in the input space.

b) Input Data Dimensionality (input_len):

It specifies the dimensionality of the input data. Each input sample should have the same dimensionality, and it should match the input_len parameter. Table III shows the dimensionality of input data in terms of the number of features obtained after feature selection for 4 considered datasets of 4 code smells. The number of features obtained after feature selection is given in Table II of Section IV.

c) Learning Rate (initial_lr):

It determines the initial learning rate for updating the weights of neurons during training. The learning rate typically decreases over time as training progresses, allowing the model to converge to a stable state gradually.

d) Neighborhood Radius (sigma):

It defines the neighborhood radius around the best-matching unit (BMU) during training. Neurons within this radius will have their weights updated during each iteration of training.

The neighborhood radius typically decreases over time as training progresses.

e) Number of Iterations (iterations):

It specifies the total number of iterations or epochs for which the SOM will be trained. Each iteration involves presenting a random input sample to the SOM and updating the weights of the neurons accordingly.

f) Topology (toroidal):

It specifies whether the SOM grid has a toroidal (circular) topology or not. Toroidal topology allows the edges of the grid to wrap around, creating a seamless map without borders.

g) Random Initialization (random_seed):

It sets the random seed used for initializing the weights of the neurons. Setting a fixed random seed ensures reproducibility of results across multiple runs.

TABLE III
PARAMETERS USED IN MINISOM

SI No.	Tuned Parameters	A brief description of the parameter	Value (s)
1	(m, n)	Grid size	LM-(97,97), FE-(90,90), GC-(90,90), DC-(90,90)
2	initial_lr	Learning rate	0.5
3	iterations	Number of Iterations	2000
4	toroidal	Topology	rectangular
5	random_seed	Random Initialization	None
6	sigma	Neighborhood Radius	0.3
7	input_len	Input Data Dimensionality	LM-18, FE-20, DC -15, GC- 15

The experiments were conducted on a standalone computer system using Keras [37] in Jupyter Notebook. For the experimentation, the study has followed the guidelines of the MiniSom [36]. The datasets under consideration were partitioned into a 70:30 ratio for training and testing purposes.

VI. PERFORMANCE MEASURES

Performance metrics namely precision, recall, F-measure, accuracy, AUC-ROC (Area Under the Receiver Operating Characteristic Curve), and MCC (Matthews Correlation Coefficient) are frequently employed in the assessment of machine learning models [8], including those for identifying code smells in software systems. The aforementioned performance measures offer valuable insights regarding the efficacy of machine learning models in identifying code smells. The following paragraphs discuss each of these measures.

1) Precision is a numerical metric that indicates the proportion of accurately detected instances of code smells out of the total number of occurrences categorized as code smells by the model.
2) Recall is a quantitative measure that evaluates the ratio of correctly identified instances of code smells to the overall number of code smells in the dataset. A high recall value indicates that the model effectively detects a substantial number of code smells.
3) F-measure is a quantitative measure used to calculate the harmonic mean of precision and recall. Accuracy is a statistical measure that calculates the proportion of accurately classified instances. In terms of code smell detection, accuracy can be defined as the measure of how correct the model's predictions are. While accuracy is commonly used as a metric, it is not appropriate in the case of imbalanced datasets with a low prevalence of code smells.
4) AUC-ROC metric measures the performance of a binary classification model. The ROC curve plots the true positive rate against the false positive rate at various thresholds. The AUC represents the area under this curve, with values ranging from 0 to 1. A higher AUC indicates better model performance, with 1.0 being perfect and 0.5 representing random guessing.
5) MCC is a quantitative metric that assesses the performance of a binary classification. It is calculated based on the confusion matrix and it ranges between -1 and 1. The value of 1 denotes a prediction that perfectly aligns with the observation, while 0 signifies a prediction made at random. On the other hand, MCC = -1 indicates complete disagreement between the prediction and the observation. An MCC score greater than 0.70 is commonly considered to be statistically significant.

VII. RESULTS AND DISCUSSIONS

This section presents the results of the undertaken study in the paper. Subsection A specifies the results when feature selection is not used. Subsection B gives the results where feature selection is employed first before applying SOM. Subsection C presents a comparison of the results of our study with the existing results of the previous study.

A. Results without feature selection

Table IV provides the results for code smell detection for 4 different code smells using six performance measures where no feature selection technique has been used. It is observed that precision is very good (> = 0.79) for all four code smells. Also, since AUC is greater than 0.70 for all considered code smells hence SOM can differentiate well between smelly and non-smelly instances. Similarly, other performance measures are also very good (> = 0.70) indicating the feasibility of code smell detection using SOM. Figure 3 below shows a bar chart representing the performance of SOM for 4 code smells.

It is observed that precision is very good (> = 0.79) for all four code smells. Also, since AUC is greater than 0.70 for all considered code smells hence SOM can differentiate well between smelly and non-smelly instances. Similarly, other performance measures are also very good (> = 0.70) indicating the feasibility of code smell detection using SOM. Figure 3 below shows a bar chart representing the performance of SOM for 4 code smells.

B. Results of the proposed method (using feature selection)

Table V below shows the results for 4 considered code smells when a technique of feature selection was employed before applying SOM. Results show that precision is much better than the precision in 7.1 for all four code smells. In this case, precision is greater than or equal to 0.89 for all 4 code smells. Also, the same is true concerning other performance measures. Table V also shows the range of performance measures, indicating that feature selection plays a significant role and enhances the overall performance of SOM for all four code smells considered in the study. Figure 4 below shows a bar chart representing the performance of SOM for 4 code smells using the proposed method.

TABLE IV
RESULTS (WITHOUT FEATURE SELECTION)

Performance measures	LM	FE	DC	GC	Range
Precision	0.84	0.88	0.79	0.94	0.79-0.94
Recall	0.78	0.58	0.93	0.75	0.58-0.93
F-measure	0.81	0.70	0.85	0.83	0.70-0.85
MCC	0.72	0.63	0.78	0.77	0.63-0.78
Accuracy	0.88	0.86	0.90	0.90	0.86-0.90
AUC	0.85	0.78	0.90	0.86	0.78-0.90

The study has used U -Matrix [38] for visualizing the output of MiniSom after its training on four datasets in figures from 5 to 8. The U-Matrix is a computational tool used to analyse the distribution of nodes in the input space based on their spacing. The U-Matrix is represented graphically as a grid cell for each node in the lattice space. The chromaticity of each node is directly proportional to the mean Euclidean distance in the input space to the neighbouring nodes of that node. The U-Matrix is a mathematical tool that can be utilized to identify clusters and outliers within a dataset.

TABLE V
RESULTS OF THE PROPOSED METHOD

Performance measures	LM	FE	DC	GC	Range
Precision	0.89	0.94	0.92	0.98	0.89-0.98
Recall	0.78	0.83	0.88	0.80	0.78-0.88
F-measure	0.83	0.88	0.90	0.88	0.83-0.90
MCC	0.76	0.84	0.85	0.83	0.76-0.85
Accuracy	0.90	0.94	0.94	0.92	0.90-0.94
AUC	0.87	0.91	0.92	0.90	0.87-0.92

In Figures 5 to 8, it is important to observe that lighter colors on the U-Matrix correspond to greater distances, while darker colors indicate denser nodes in that specific area. Observing the U-Matrix depicted in Figures 5 to 8, it is evident that there exists a significant gap between the dark color region and the light color region. This observation suggests that the dark color data points exhibit a significant distance from the light color data points. Thus, the SOM output helps to verify the map's fidelity to the underlying data.

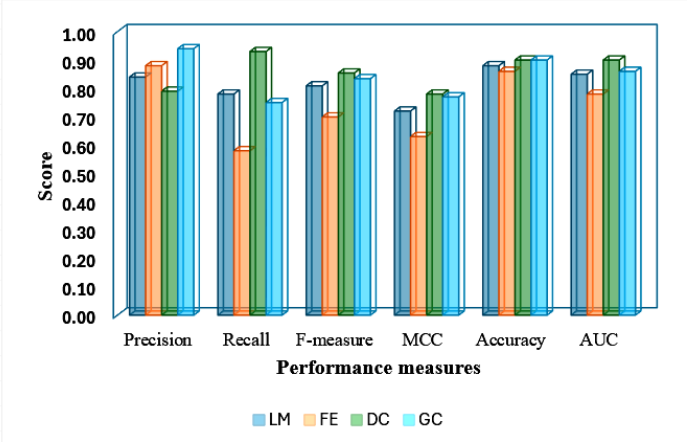

Fig. 3. Results of SOM (without feature selection)

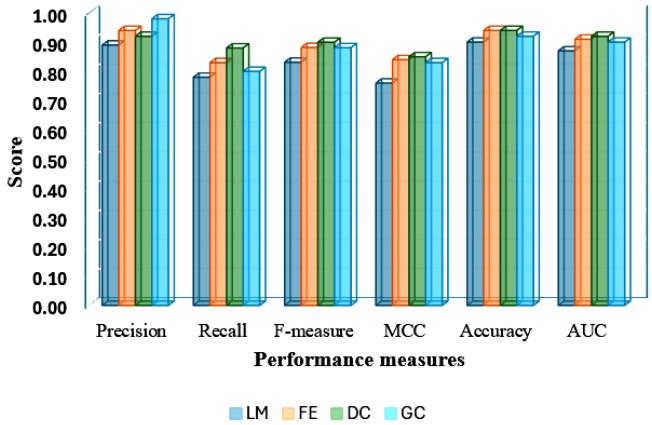

Fig.4. Results of the proposed method

C. Comparison with the existing study

The previous study by Fontana [9] used 16 different machine learning algorithms to detect 4 code smells using 4 different datasets. The results in this study [9] have been shown in terms of performance measures such as accuracy, F-measure, and AUC. It is observed that algorithm B -J48 Pruned produced the best results in terms of all performance measures. Similarly, for god class, naïve Bayes gave the best results for 3 considered performance measures. For long method and feature envy, B -J48 Pruned gave the best results in terms of all performance measures. Table VI presents the results for the 3 performance measures. Also, a symbol dash (-) indicates that the scores are not available in the study by Fontana [9] .

To contrast the results of two approaches (the proposed approach and the approach of supervised machine learning algorithms used in the previous study by Fontana [9]) and to ascertain the superior approach, a statistical test was employed. The selected statistical test for comparison is the Wilcoxon signed rank test as outlined by Hollander et al. [39]. The test is non-parametric and allows for pairwise comparison of configuration parameters of the algorithms.

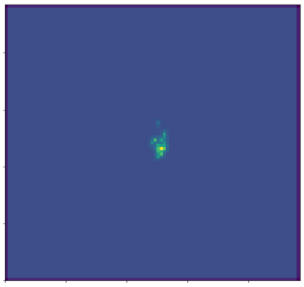

Fig. 5. U matrix for LM

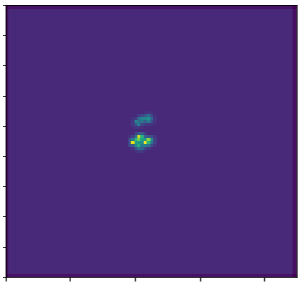

Fig. 6. U matrix for FE

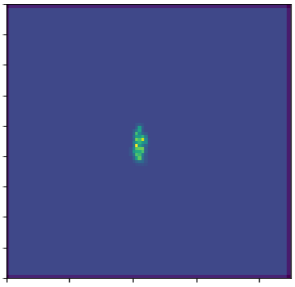

Fig. 7. U matrix for DC

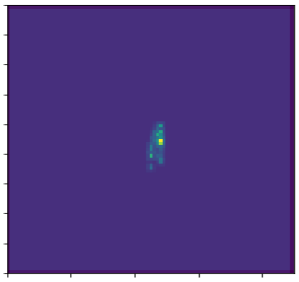

Fig. 8. U matrix for GC

TABLE VI
COMPARISON OF THE PROPOSED METHOD WITH THE PREVIOUS STUDY

	The proposed approach				Previous study (SML algorithms) [9]			
Performance measures	LM	FE	DC	GC	LM	FE	DC	GC
Precision	0.89	0.94	0.92	0.98	-	-	-	-
Recall	0.78	0.83	0.88	0.80	-	-	-	-
MCC	0.76	0.84	0.85	0.83	-	-	-	-
Accuracy	0.90	0.94	0.94	0.92	0.99	0.97	0.99	0.97
AUC	0.87	0.91	0.92	0.90	0.99	0.99	0.99	0.99
F-measure	0.83	0.88	0.90	0.88	0.99	0.97	0.99	0.98

This method can be used to compare the performance of two algorithms to ascertain their equality or determine which one performs better. When the test results show that one algorithm has better performance than another, the former is labeled as the winner and the latter as the loser. When the test results show that the two performances are indistinguishable, it means that the algorithms do not demonstrate a clear advantage or disadvantage.

The test was carried out with a significance level set at 0.05 using the 2-tailed test. Appendix D displays the data and the results of the Wilcoxon signed rank test performed, considering all 4 code smells together due to the limited number of samples available for each code smell. The results demonstrate that the SML algorithms used in the previous study were superior to the proposed approach. Nevertheless, the results of the proposed approach correspond substantially with those of the previous study, suggesting that the proposed approach has the potential to identify code smells through UnML algorithm(s). Therefore, the presented study encourages further investigation by researchers.

D. Threats to validity

Lastly, we consider the possible threats that could undermine the legitimacy of our study, as outlined by Runeson [40]. Based on Runeson's research, legitimacy can be threatened in four distinct categories: external threats, construct threats, reliability threats, and internal validity threats. External validity encompasses several potential issues that could undermine the overall validity of the proposed method and its outcomes. The purpose of internal validity is to identify and establish causal links, as well as confirm the connection between variables and logical results. The study does not currently consider construct validity and reliability due to their lack of applicability.

D.1 External Validity

Ultimately, two potential threats have been identified. The study was limited to open-source projects of Java; therefore, the results cannot be considered useful to other programming languages. Moreover, it is crucial to acknowledge that the study solely concentrated on open-source Java projects. Consequently, the generalizability of the results to industrial software necessitates verification through the study's findings.

D.2 Internal Validity

The factors that influence the results are often internal validity threats. The initial factor pertains to the specific UnML technique employed in the study. The study has used a single technique: SOM for the experimentation but other techniques may be further explored. Another contributing factor is the study's use of 4 publicly available datasets of 4 different code smells. Participating in a broader range of datasets would surely enhance the feasibility and outcome of the research study. The third factor is the choice of feature selection technique used in the study. The study has used an existing Python package Featurewiz, but other feature selection techniques may be employed.

VIII. CONCLUSION & FUTURE WORK

The study has come up with a proposed method that utilizes a Self-Organizing Map (SOM), an UnML algorithm. The proposed method was validated on four popular and publicly available datasets of four different code smells such as long method, feature envy, god class, and data class. The performance was evaluated on several performance measures namely AUC, precision, recall, F-measure, accuracy, and MCC. The results showed that the proposed method effectively detected all 4 code smells with high scores for precision (0.89-0.98), recall (0.78-0.88), accuracy (0.90-0.94), and AUC (0.87-0.92). The results of the proposed method were evaluated against a prior study that utilized supervised

machine learning algorithms. The comparison revealed that the results of the proposed method closely aligned with those of the previous study. Therefore, the proposed method shows significant potential in identifying code smells.

UnML methods provide valuable advantages for code smell detection. They can identify patterns by detecting common code smells by analyzing patterns in code repositories without labeled examples. They can offer insights into codebase structure, aiding developers in understanding code health and areas needing improvement. They can do feature selection by automatically extracting meaningful representations of code, aiding in the detection of code smells. They can flag anomalous code patterns, highlighting potential instances of code that smells like dead code or inconsistent naming conventions. They provide objective assessments of code quality by learning directly from the data distribution, helping prioritize refactoring efforts based on data-driven insights.

Future scope- Exploration of a broader spectrum of open source and industrial Java projects will be incorporated into future research. Additionally, the simultaneous detection of multiple code smells is another potential area for future consideration. The recommendation of software metrics crucial to detecting code smells using UnML could be investigated in future studies.

REFERENCES

[1] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts: "*Refactoring: Improving the Design of Existing Code*". 2002.

[2] T. Sharma and D. Spinellis: *A survey on software smells*, J. Syst. Softw., vol. 138, pp. 158–173, 2018.

[3] L. Da Silva Sousa: *Spotting design problems with smell agglomerations*, in Proceedings - International Conference on Software Engineering, May 2016, pp. 863–866.

[4] Y. A. Khan and M. El-Attar: *Using model transformation to refactor use case models based on antipatterns*, Inf. Syst. Front., vol. 18, no. 1, pp. 171–204, Aug. 2016.

[5] T. Sharma, M. Fragkoulis, and D. Spinellis: *Does your configuration code smell?*, in Proceedings - 13th Working Conference on Mining Software Repositories, MSR 2016, May 2016, pp. 189–200.

[6] T. Sharma and D. Spinellis: *A survey on software smells*, J. Syst. Softw., vol. 138, no. December 2017, pp. 158–173, 2018.

[7] M. I. Azeem, F. Palomba, L. Shi, and Q. Wang: *Machine learning techniques for code smell detection: A systematic literature review and meta-analysis*, Inf. Softw. Technol., vol. 108, no. 4, pp. 115–138, 2019.

[8] A. Al-Shaaby, H. Aljamaan, and M. Alshayeb: *Bad Smell Detection Using Machine Learning Techniques: A Systematic Literature Review*, Arab. J. Sci. Eng., no. 0123456789, 2020.

[9] F. Arcelli Fontana, M. V. Mäntylä, M. Zanoni, and A. Marino: *Comparing and experimenting machine learning techniques for code smell detection*, Empir. Softw. Eng., vol. 21, no. 3, pp. 1143–1191, 2016.

[10] J. Kreimer: *Adaptive detection of design flaws*, Electron. Notes Theor. Comput. Sci., vol. 141, no. 4 SPEC. ISS., pp. 117–136, 2005.

[11] F. Khomh, S. Vaucher, Y. G. Guéhéeneuc, and H. Sahraoui: *A bayesian approach for the detection of code and design smells*, in Proceedings - International Conference on Quality Software, 2009, pp. 305–314.

[12] S. Bryton, F. Brito E Abreu, and M. Monteiro: *Reducing subjectivity in code smells detection: Experimenting with the Long Method*, in Proceedings - 7th International Conference on the Quality of Information and Communications Technology, QUATIC 2010, 2010, pp. 337–342.

[13] F. Khomh, S. Vaucher, Y. G. Guéhéneuc, and H. Sahraoui: *BDTEX: A GQM-based Bayesian approach for the detection of antipatterns*, J. Syst. Softw., vol. 84, no. 4, pp. 559–572, 2011.

[14] Svmd. Maiga *et al.: Support Vector Machines for Anti-pattern Detection*, 2012, Accessed: Feb. 17, 2020. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/6494935/.

[15] D. K. Kim: *Finding bad code smells with neural network models*, Int. J. Electr. Comput. Eng., vol. 7, no. 6, pp. 3613–3621, 2017.

[16] M. Hadj-Kacem and N. Bouassida: *A hybrid approach to detect code smells using deep learning*, ENASE 2018 - Proc. 13th Int. Conf. Eval. Nov. Approaches to Softw. Eng., vol. 2018-March, no. Enase 2018, pp. 137–146, 2018.

[17] H. Liu, Z. Xu, and Y. Zou: *Deep learning based feature envy detection*, Proc. 33rd ACM/IEEE Int. Conf. Autom. Softw. Eng. - ASE 2018, pp. 385–396, 2018.

[18] A. Barbez, F. Khomh, and Y. G. Guéhéneuc: *A machine-learning based ensemble method for anti-patterns detection*, J. Syst. Softw., vol. 161, 2020.

[19] M. Gupta: *A Novel Approach for Code Smell Detection : An Empirical Study*, IEEE Access, vol. 9, pp. 162869–162883, 2021.

[20] N. Vatanapakorn, C. Soomlek, and P. Seresangtakul: *Python Code Smell Detection Using Machine Learning*, ICSEC 2022 - Int. Comput. Sci. Eng. Conf. 2022, no. April 2023, pp. 128–133, 2022.

[21] T. Sharma, V. Efstathiou, P. Louridas, and D. Spinellis: *Code smell detection by deep direct-learning and transfer-learning*, J. Syst. Softw., vol. 176, p. 110936, 2021.

[22] T. Sharma: *On the Feasibility of Transfer-learning Code Smells using Deep Learning*, ACM Trans. Softw. Eng. Methodol., vol. 1, no. 1, pp. 1281–1284, 2019.

[23] R. Gupta and S. K. Singh: *A Novel Transfer Learning Method for Code Smell Detection on Heterogeneous Data: A Feasibility Study*, SN Comput. Sci., vol. 4, no. 6, pp. 1–21, Nov. 2023.

[24] M. Long, J. Wang, J. Sun, and P. S. Yu: *Domain invariant transfer kernel learning*, IEEE Trans. Knowl. Data Eng., vol. 27, no. 6, pp. 1519–1532, 2015.

[25] R. Sandouka and H. Aljamaan: *Python code smells detection using conventional machine learning models*, PeerJ Comput. Sci., vol. 9, 2023.

[26] L. Madeyski and T. Lewowski: *Detecting code smells using industry-relevant data*, Inf. Softw. Technol., vol. 155, 2023.

[27] "Featurewitz · PyPI." https://pypi.org/project/featurewiz/ (accessed Jun. 02, 2023).

[28] A. AbuHassan, M. Alshayeb, and L. Ghouti: *Software smell detection techniques: A systematic literature review*, J. Softw. Evol. Process, no. September 2019, pp. 1–48, 2020.

[29] E. Tempero *et al.: The Qualitas Corpus: A curated collection of Java code for empirical studies*, in Proceedings - Asia-Pacific Software Engineering Conference, APSEC, 2010, pp. 336–345, doi: 10.1109/APSEC.2010.46.

[30] N. Petrov, A. Georgieva, and I. Jordanov: *Self-organizing maps for texture classification*, Neural Comput. Appl., vol. 22, no. 7–8, pp. 1499–1508, 2013.

[31] X. Qu *et al.: A Survey on the Development of Self-Organizing Maps for Unsupervised Intrusion Detection*, Mob. Networks Appl., vol. 26, no. 2, pp. 808–829, 2021.

[32] I. Jammoussi and M. Ben Nasr: *A hybrid method based on extreme learning machine and self organizing map for pattern classification*, Comput. Intell. Neurosci., vol. 2020, 2020.

[33] D. Pratiwi: "The Use of Self Organizing Map Method and Feature Selection in Image Database Classification System," 2012, [Online]. Available: http://arxiv.org/abs/1206.0104.

[34] U. Fidan, N. Ozkan, and I. Calikusu: *Clustering and classification of dermatologic data with Self Organization Map (SOM) method*, in 2016 Medical Technologies National Conference, TIPTEKNO 2016, 2017, no. June, pp. 1–4, doi: 10.1109/TIPTEKNO.2016.7863075.

[35] T. Li, G. Sun, C. Yang, K. Liang, S. Ma, and L. Huang: *Using self-organizing map for coastal water quality classification: Towards a better understanding of patterns and processes*, Sci. Total Environ., vol. 628–629, pp. 1446–1459, 2018.

[36] "GitHub - JustGlowing/minisom: MiniSom is a minimalistic implementation of the Self Organizing Maps." https://github.com/JustGlowing/minisom (accessed Mar. 13, 2024).

[37] "Keras: The high-level API for TensorFlow." https://www.tensorflow.org/guide/keras.

[38] A. Ultsch: "U * Matrix : a Tool to visualize Clusters in high dimensional Data," 2014.

[39] E. C. Hollander, Myles, Douglas A. Wolfe: "Nonparametric statistical methods", vol. 102. 2013.

[40] P. Runeson, M. Höst, A. Rainer, and B. Regnell: "Case Study Research in Software Engineering", 2012.

APPENDIX

https://docs.google.com/document/d/1POrGZwV1ZpWiiMr8rXpYYWf1kXqHxgad/edit?usp=sharing&ouid=102221464989892511063&rtpof=true&sd=true

Ruchin Gupta is an accomplished academic and researcher with over 23 years of experience, currently serving as an Assistant Professor at KIET Group of Institutions, Delhi-NCR, Ghaziabad. He is an active member of several prestigious engineering societies and has contributed extensively to the field through his roles as a reviewer for top-tier journals and as a technical chair at international conferences. His research spans machine learning, code smell detection, and blockchain technology, with numerous publications in high-impact journals. Ruchin has also been recognized with awards for his scholarly contributions and holds patents in AI, ML, and IoT.

Dr. Narendra Kumar is working as Dean-Incubation and professor in CSE department, GCET. He has worked as co-founder and President at Innotekverse Pvt Limited (Vorphy). He has more than 18 years of national and international academic experience and 3 years of experience in corporate. He has done BTech (IT), MTech (CSE), and Ph.D. in Computer Science and Engineering (CSE). He has been an entrepreneur also and has established many incubation centers. He has been the team lead for many E-Learning projects. His current domain is Blockchain and XR technologies. He has been in leading positions in many universities and handled international accreditations. He has established many centers of excellence in the latest technologies. He is IBM certified Blockchain Developer.

Dr. Sunil Kumar is working as an Associate Professor in Gagotias College of Engineering and Technology Greater Noida, Uttar Pradesh. He has more than 17 years of teaching experience. He pursued his Ph.D degree (2022) in Computer Science and Engineering from SRM Institute of Science and Technology, Delhi NCR Campus Modinagar, Ghaziabad, Uttar Pradesh and master's degree in computer engineering from Shobhit University Meerut in 2012, and received the bachelor's degree in information technology from Krishna Institute of Engineering and Technology, Ghaziabad, U.P. in 2006,. He has published more than 20 research papers in journals and conferences. His fields of interest are Artificial Intelligence, Machine Learning and Computer Networks. He is a member of IAENG, IEEE and ISTE professional societies.

Dr. Jitendra Kumar Seth is working as an Associate Professor in KIET Group of Institutions Ghaziabad. He has completed his PhD degree from Jaypee Institute of Information Technology, Noida in 2019. He obtained his M.Tech degree from Shobhit University Meerut in 2009 and B.Tech degree from Radha Govind Engineering College Meerut in 2004. During his teaching career of more than 19 years, he has taught courses such as Cloud Computing, Big Data, Distributed System, OOP, Web Technology, Principle of programming languages etc. Dr. Jitendra has participated in many National and International conferences and has published more than 20 research papers in reputed International Journals and Conferences. He has guided more than 50 graduate projects and supervising PhD scholars. He has expertise on programming languages like Java, Python, PHP and XML. His research area includes Cloud Computing, Big Data, Cyber Security, Machine Learning and Search Engines.