

# Code Smell Detection Using Classification Approaches



Seema Dewangan and Rajwant Singh Rao

**Abstract** Code smells combine software features that indicate a code or design-related problem that can increase the difficulty of the software and make it complicated to recognize, develop, and maintain the software. Code smell detection helps us to make code accurate and effective. In this study, five classification models are applied. To complete this study, four datasets of code smell are used. To pick the finest feature in the existing dataset, a feature selection method (FST) is used. The ten-iteration cross-validation method is applied to calculate the execution of the model predicted. In this document, the random forest model for the Feature-envy dataset achieves an accuracy of 0.9912%.

**Keywords** Classification methods · Feature selection technique (FST) · Code smell detection

## 1 Introduction

The software complexity is continuously mounting because of the complex requirements, an increase in the quantity, module size, and code smells belonging to the advanced software, etc. Harsh conditions are challenging to examine and recognize, and therefore, as a result the improvement turns out to be problematic. The complicated necessities are not in the hands of the designer, but the smell of the code can be recognized and the software can be made modest, more comprehensible, and uncomplicated [1]. During the software making procedure, the operational and non-operational quality required for developers must be followed to secure software quality [2]. Developers concentrate only on practical demands and ignore non-functional needs, such as comprehensibility, verifiability, evolution, maintainability, and reusability [3]. The complexity of the software program is growing constantly due to the wide range of module sizes, method sizes, and branching complexity, renovation costs are increasing due to extra code smells, and the quality of the software

---

S. Dewangan (✉) · R. S. Rao  
Guru Ghasidas Vishwavidyalaya (A Central University), Bilaspur, C.G., India  
e-mail: [sskd501@gmail.com](mailto:sskd501@gmail.com)

program is deteriorating due to the growing number of lines of code. Fowler et al. [4] described the refectory method from which the slackly executed program could be changed in an exemplary execution; 22 definitions of code smells were given by them.

Many approaches for code smell detection have been used in the literature to identify smells of different codes.

Every method will produce a distinct outcome according to their different categories. Seven types of code smell detection techniques are categorized by Kessentini et al. [5]. These seven techniques are cooperative-based technique [6], visualization-based technique [7], search-based method [8], probabilistic technique [9], symptom-based technique, metrics-based technique, and manual technique [10]. Inspection approaches [11], manufacturing process, and process identification methods [12] were used by manual technique to advance the superiority of software. To detect code smells using a symptom-based technique, the specification algorithm was used [13].

Multiple studies have analyzed the code smell's influence at the software and also displayed unwanted impacts of software's features with code smell detection [14, 15]. They also analyze the code smell effects that increase the risk of software system failures and faults. They found that the challenge full of code smell impacted the software improvement procedure and suggested software's refactoring for elimination of it.

Deligiannis et al. [16], Olbrich et al. [17, 18], and Khomh et al. [19] observed the effect of code smells on software development by observing the changes in the occurrence and size in the software system. They also observe that classes affected by code smells have a more significant rate of change and require supplementary maintenance work. The relevance of bad smells and the possibility of class error in an OOS system were investigated by Li and Shatnawi [20]. Infected software elements that use code smells have more class mistakes than other elements, according to the experiment. The negative effect of God-class on energy intake was examined by Perez-Castillo and Piattini [21], who found that eliminating God-class odors reduces the cyclomatic complexity of the source code in the software system.

The main contributions of this research work are divided into two parts: In the first part, five classification techniques are applied to detect the code smell from the dataset and feature selection technique is also applied to select the best features from each dataset. The second part shows the performance measures obtained using classification and evaluation techniques with the tenfold cross-validation technique.

In this research work, the classification techniques for code smell detection are proposed. The four datasets of code smell datasets are considered. The class-level smell contains the God-class and the Data-class datasets, whereas method-level smell contains Feature-envy and Long-method datasets. Five classification methods (random forest, SVM, naive Bayes, KNN, and logistic regression) are applied to classify the dataset.

This paper is partitioned in five section: Sect. 2 describes the literature review, and this section shortly reviews the work done in past by various authors for code

smell discovery by classification techniques. Section 3 describes proposed methodology. Section 4 describes the investigational consequences, and Sect. 5 explains the conclusion of our work.

## 2 Related Work

Many researchers have presented papers using ML algorithms to detect the code smells. In this paper, the existing methods of supervised learning techniques are used to detect code smells. In this paper, the existing methods of supervised learning techniques are used to detect code smells to comparing and experimenting with ML algorithms to detect code smells. They tested sixteen ML techniques on four code smell datasets as well seventy four Java platforms on the training dataset that were manually evaluated. Boosting approaches are also used on four datasets of code smells.

Mhawish and Gupta [22–24] proposed software metrics, tree-based and decision tree-based ML algorithms, and software metrics for differentiating and recognizing similar structural design patterns. To choose the most significant characteristics from each dataset, they utilized two feature selection strategies in light of GA-CFS (genetic algorithm).

They also employed a parameter refinement using a grid search method approach to improve the accurateness of all machine learning methods. Guggulothu and Moiz [25, 26] suggested a multi-label classification strategy for code smell detection. To see if the specified code components are affected in several ways, they employed a categorization system with many labels. For excellent accuracy, they made use of an unsupervised classification algorithm. Dewangan et al. [27] applied six ML algorithms, and two feature selection techniques such as chi-square and method for selecting features based on a wrapper were applied to pick the greatest features from each dataset; then, moreover grid search procedure was used to increase the performance of model, and they obtained 100% highest accuracy using the logistic regression technique for the Long-method dataset. Kreimer [28] proposed a detection approach to detect Long-methods and prominent class code smell based on a decision tree approach. The approach is evaluated on two small software: the WEKA software package and the IYC system. It was found that the prediction model and this model help detect code smells. The usefulness of decision trees for identifying code odors was proposed by Amorim et al. [29]. By putting Kreimer’s decision tree model to the test, they were able to corroborate his findings. Class change proneness can be predicted based on code smell using Pritam et al.’s [30] machine learning methods. They agree that code smells have an influence on the predisposition of a given session in a produce context to change. They used six ML techniques to estimate variation proneness based on code smells from 8200 Java modules across 14 software systems.

Draz et al. [31] proposed employing the classifier based on the whale optimization method to enhance code smell prediction using a search-based method. They tested

**Table 1** Summary of related work

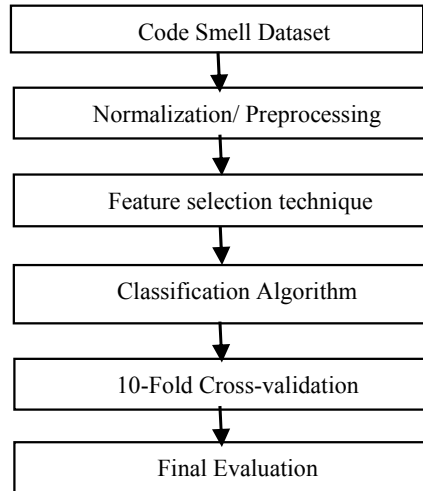
Author name	Model	Datasets	FST	Results
Fontana et al. [33]	16 machine learning algorithm	Code smell datasets	Non	99.10% in B-J48 pruned on Long-method
Guggulothu and Moiz [25]	B-random forest, random forest, J48 unpruned etc.	Feature-envy and Long-method with multi-label classification	Non	95.9% in random forest on Long-method, 99.1% in B-J48 pruned on Feature-envy
Mhawish and Gupta [23]	Decision tree algorithm	Code smell dataset	Genetic algorithm-based two FST	Data-class—98.05%, God-class—97.56, and Long-method—98.38%
Mhawish and Gupta [22]	Six machine learning algorithm	Code smell dataset	Genetic algorithm-based two FST	99.70% in Data-class using random forest model

the nine different kinds of code smells on five different open systems applications. They had an accuracy of 94.24% and a recall of 93.4% on average.

For metric-based code smell detection, Pecorelli et al. [32] provided an interesting finding comparing the performance of machine learning-based and heuristic-based strategies. They considered five types of code smells (God-class, Spaghetti Code, Class Data Should Be Private, Complex Class, and Long-method) and compared ML techniques with DECOR, a state-of-the-art heuristic-based approach. Researchers discovered that the DECOR consistently outperformed the ML baseline. In Table 1, a summary table of some essential related work is shown.

### 3 Proposed Methodology

In this paper, a code smell detection framework is constructed using classification models. Code smell matrices play an essential part in determining the operational as well as non-operational abilities and recognizing the software's properties. Metrics manage the static information of the software, such as classes, methods, and parameters that measure coupling and cohesion between objects in the system. Figure 1 depicts the steps which are followed to build the code smell detection framework. First, four datasets of code smell are created. The pretreatment (regularization) processes are then done to the dataset to cover all of the dataset's ranges. The best features from each dataset are then selected using the wrapper-based FST. Then train the model with classification algorithms applied to the dataset and determine their performance. The methodology of tenfold cross-validation is then applied to assess the result of each experiment during the preparation development. For performance

**Fig. 1** Proposed work

measurement, tenfold cross-validation is used, which divides the dataset into ten sections and repeats them ten times. Then evaluate the final results.

God-class, Data-class, Feature-envy, and Long-method are four code smell datasets which [33] were taken to make the code smell detection framework in this study. In the following section, the data preparation methodology is shown briefly.

Because various datasets have distinct attributes, we cannot always use straightforward classification techniques on them. As a result, normalization is required to span the dataset's various ranges. ML models may sometimes evaluate quickly on a normalized dataset, which might have a big impact when the model is sensitive to size. Prior to the implementation of the support vector machine algorithm, for example, it is necessary to avoid normalization in order to dominate higher number ranges on small number varieties, where the variety of possible elevated values causes mathematical problems [34]. This article uses the minimum–maximum normalization technique to convert dataset values between 0 and 1. This strategy is utilized in the data preparation step, which prepares the data for subsequent processing using one of the machine learning algorithms such as SVM, NN, and others [35]. The following equation executes an  $x$  mapping change from feature  $A$  from the range  $[\min A, \max A]$  to  $[\text{new min } A, \text{new max } A]$ .

$$x' = (x - \min A) / (\max A - \min A)$$

All datasets were subjected to the min–max normalization approach, and the resulting new data was used as input into all classification systems.

To choose the best features (matrices) from each dataset, this experiment uses a wrapper-based FST. FST is applied to choose a set of characteristics in the dataset that are mainly appropriate to the goal value [36]. In this experiment, we have selected

the ten best features from each dataset, and then, classification algorithms are applied to each dataset.

This paper applies five classification algorithms (random forest, SVM, naive Bayes, KNN, and logistic regression) to perceive code smells from the code smell dataset. Classification algorithms classify the data into the specified number of classes in our dataset.

In this study, the validation technique is used to assess the performance of each experiment. For this, a tenfold cross-validation training approach was applied. Classification models that partition the dataset into tenfold with ten times of iteration are calculated using tenfold cross-validation. Different parts of the dataset are considered test datasets at each iteration, and other convolutions of the dataset are considered training modes. Then, finally the trained models are tested with unseen test dataset (10% split from the dataset before training). Stealth test dataset is used to clarify the model's forecasts and escape making broad generalizations.

Four performance constraints, precision, recall,  $F1$ -score, and accuracy, were examined to measure the efficiency of our classification approach. To calculate them, TP, TN, FP, and FN are found through the confusion matrix. True positive (TP) displays the occurrences in the positive class that properly forecast the model. False positive (FP) refers to occasions in which the model is predicted inaccurately in the positive class. True negative (TN) displays the instances in the negative class that properly forecast the model. Furthermore, false negative (FN) displays situations where the negative class is wrongly predicted.

## 4 Experimental Results

To the experiment work, four code smell datasets are used. The five classification algorithms are applied to identify the code smells from each dataset. The four performance measurements, precision ( $P$ ), recall ( $R$ ),  $F1$ -score ( $F1$ ), and accuracy ( $A$ ), are considered for each dataset. The experimental results for each classification technique are shown in Table 2. In this research, the  $F1$ -score was 0.98%, and accuracy was 0.98% for the Data-class, the  $F1$ -score was 0.98%, and the accuracy was 0.97% for the God-class, the  $F1$ -score was 0.98%, and an accuracy of 0.9912% for the Feature-envy, and an  $F1$ -score of 1.00% and an accuracy of 0.9952% for the Long-method using the random forest algorithm attained the maximum accuracy, whereas the naive Bayes (0.91% accuracy for Feature-envy) attained the gives the worst performance.

### 4.1 Evaluation of Our Techniques to Other Related Works

Table 3 represents a brief long evaluation of our techniques with other related works. In this evaluation, it is observed that in the Feature-envy dataset our approach achieved 99.12% accuracy, while in the Data-class and God-class datasets, Mhawish

**Table 2** Experimental result of five classification techniques with four code smell datasets

	Data-class				God-class				Feature-envy				Long-method			
	<i>P</i>	<i>R</i>	<i>F-1</i>	<i>A</i>	<i>P</i>	<i>R</i>	<i>F-1</i>	<i>A</i>	<i>P</i>	<i>R</i>	<i>F-1</i>	<i>A</i>	<i>P</i>	<i>R</i>	<i>F-1</i>	<i>A</i>
	Random forest	0.97	0.96	0.98	0.98	0.97	0.98	0.98	0.97	1.00	0.95	0.98	0.9912	1.00	1.00	1.00
SVM	1.00	1.00	1.00	0.97	0.98	0.99	0.99	0.96	0.99	0.94	0.96	0.94	1.00	1.00	1.00	0.97
Naive Bayes	0.90	0.93	0.91	0.92	0.97	0.98	0.98	0.96	1.00	1.00	1.00	0.91	0.82	0.95	0.86	0.94
KNN	0.98	0.99	0.99	0.95	0.99	0.98	0.99	0.96	0.97	0.90	0.93	0.92	1.00	1.00	1.00	0.98
Logistic regression	0.98	1.00	0.99	0.96	1.00	0.98	0.99	0.96	1.00	0.97	0.99	0.96	1.00	1.00	1.00	0.98

*P*: Precision, *R*: Recall, *F-1*: *F1*-score, *A*: Accuracy

**Table 3** Comparison of our approach with other related work

Year	Author name	Datasets			
		Data-class	God-class	Feature-envy	Long-method
		Accuracy (%)	Accuracy (%)	Accuracy (%)	Accuracy (%)
2016	Fontana et al. [33]	99.02	97.55	96.94	99.43
2020	Mhawish and Gupta [22]	99.70	98.48	97.97	95.97
2020	Guggulothu and Moiz [25]	N/A	N/A	99.10	95.90
2021	Dewangan et al. [27]	99.74	98.21	98.60	100.00
2021	Our approach	98.00	97.00	<b>99.12</b>	99.52

Bold value represent that our approach obtained the highest accuracy (99.12%) for feature-envy dataset in comparison to other authors

and Gupta [22] achieved 99.70 and 98.48% highest accuracy. For the Long-method data set, Dewangan et al. [27] achieved the highest accuracy of 100%.

## 5 Conclusion and Next Steps

The classification strategy is provided in this research to identify the code smells from software and to find the metrics that play an important part in the detection process using classification algorithms. To determine the key metrics that may be utilized to increase accuracy, the wrapper-based feature selection approach is used. The findings are then evaluated using a tenfold cross-validation procedure. In this research work, it has been noted that the random forest procedure achieved the maximum accuracy of 0.98% for Data-class, 0.97% for God-class, 0.9912% for Feature-envy, and 0.9952% for Long-method dataset. In the future work, other machine learning techniques and other metrics selection techniques can be applied to increase the outcomes.

## References

1. Lehman MM (1980) Programs, life cycles, and laws of software evolution. Proc IEEE 68(9):1060–1076
2. Wiegers K, Beatty J (2013) Software requirements. Pearson Education
3. Chung L, do Prado Leite JCS (2009) On non-functional requirements in software engineering. In: Mylopoulos J, Borgida AT, Chaudhri V, Giorgini P, Yu E (eds) Conceptual modeling: foundations and applications-essays. Springer, pp 363–379
4. Fowler M, Beck K, Brant J, Opdyke W, Roberts D (1999) Refactoring: improving the design of existing code, 1st edn. Addison-Wesley Professional



5. Kessentini W, Kessentini M, Sahraoui H, Bechikh S, Ouni A (2014) A cooperative parallel search-based software engineering approach for code-smells detection. *IEEE Trans Softw Eng* 40(9):841–861
6. Abdelmoez W, Kosba E, Iesa AF (2014) Risk-based code smells detection tool. In: The international conference on computing technology and information management (ICCTIM2014). The Society of Digital Information and Wireless Communication, pp 148–159
7. Murphy-Hill E, Black AP (2010) An interactive ambient visualization for code smells. In: Proceedings of the 5th international symposium on software visualization. ACM, pp 5–14
8. Palomba F, Bavota G, Di Penta M, Oliveto R, Poshyvanyk D, De Lucia A (2015) Mining version histories for detecting code smells. *IEEE Trans Softw Eng* 41(5):462–489
9. Rao AA, Reddy KN (2007) Detecting bad smells in object oriented design using design change propagation probability matrix 1
10. Marinescu R (2004) Detection strategies: metrics-based rules for detecting design AWS. In: 20th IEEE international conference on software maintenance, 2004. Proceedings. IEEE, pp 350–359
11. Travassos G, Shull F, Fredericks M, Basili VR (1999) Detecting defects in object-oriented designs: using reading techniques to increase software quality. *ACM Sigplan Not* 34:47–56
12. Ciupke O (1999) Automatic detection of design problems in object-oriented reengineering. In: Technology of object-oriented languages and systems, 1999. TOOLS 30 proceedings. IEEE, pp 18–32
13. Moha N, Gueheneuc YG, Le Meur AF, Duchien L, Tiberghien A (2010) From a domain analysis to the specification and detection of code and design smells. *Form Asp Comput* 22(3–4):345–361
14. Yamashita A, Moonen L (2013) Exploring the impact of inter smell relations on software maintainability: an empirical study. In: Proceedings of the 35th international conference on software engineering, May 2013, pp 682–691
15. Yamashita A, Counsell S (2013) Code smells as system-level indicators of maintainability: an empirical study. *J Syst Softw* 86(10):2639–2653
16. Deligiannis I, Stamelos I, Angelis L, Roumeliotis M, Shepperd M (2004) A controlled experiment investigation of an object oriented design heuristic for maintainability. *J Syst Softw* 72(2):129–143
17. Olbrich S, Cruzes DS, Basili V, Zazworka N (2009) The evolution and impact of code smells: a case study of two open source systems. In: Proceedings of the 3rd international symposium on empirical software engineering and measurement, Oct 2009, pp 390–400
18. Olbrich SM, Cruzes DS, Sjøberg DIK (2010) Are all code smells harmful? A study of god classes and brain classes in the evolution of three open source systems. In: Proceedings of the 26th IEEE international conference on software maintenance, Sept 2010
19. Khomh F, Penta DM, Gueheneuc YG (2009) An exploratory study of the impact of code smells on software change proneness. In: Proceedings of the 16th working conference on reverse engineering, Oct 2009, pp 75–84
20. Li W, Shatnawi R (2007) An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution. *J Syst Softw* 80(7):1120–1128
21. Perez-Castillo R, Piattini M (2014) Analyzing the harmful effect of god class refactoring on power consumption. *IEEE Softw* 31(3):48–55
22. Mhawish MY, Gupta M (2020) Predicting code smells and analysis of predictions: using machine learning techniques and software metrics. *J Comput Sci Technol* 35(6):1428–1445. <https://doi.org/10.1007/s11390-020-0323-7>
23. Mhawish MY, Gupta M (2020) Software metrics and tree-based machine learning algorithms for distinguishing and detecting similar structure design patterns. *SN Appl Sci* 2:11
24. Mhawish MY, Gupta M (2019) Generating code-smell prediction rules using decision tree algorithm and software metrics. *Int J Comput Sci Eng (IJCSE)* 7(5):41–48
25. Guggulothu T, Moiz SA (2020) Code smell detection using multi-label classification approach. *Softw Qual J* 28:1063–1086. <https://doi.org/10.1007/s11219-020-09498-y>

26. Guggulothu T, Moiz SA (2019) Detection of shotgun surgery and message chain code smells using machine learning techniques. *Int J Rough Sets Data Anal (IJRSDA)* 6(2):34–50. <https://doi.org/10.4018/IJRSDA.2019040103>
27. Dewangan S, Rao RS, Mishra A, Gupta M (2021) A novel approach for code smell detection: an empirical study. *IEEE Access* 9:162869–162883. <https://doi.org/10.1109/ACCESS.2021.3133810>
28. Kreimer J (2005) Adaptive detection of design flaws. *Electron Notes Theor Comput Sci* 141(4):117–136
29. Amorim L, Costa E, Antunes N, Fonseca B, Ribeiro M (2015) Experience report: evaluating the effectiveness of decision trees for detecting code smells. In: *Proceedings of the 26th IEEE international symposium on software reliability engineering*, Nov 2015, pp 261–269
30. Pritam N, Khari M, Son LH, Kumar R, Jha S, Priyadarshini I, Abdel-Basset M, Long HV (2019) Assessment of code smell for predicting class change proneness using machine learning. *IEEE Access* 7:37414–37425
31. Draz MM, Farhan MS, Abdulkader SN, Gafar MG (2021) Code smell detection using whale optimization algorithm. *Comput Mater Contin* 68(2):1919–1935
32. Pecorelli F, Palomba F, Di Nucci D, De Lucia A (2019) Comparing heuristic and machine learning approaches for metric-based code smell detection. In: *2019 IEEE/ACM 27th international conference on program comprehension (ICPC)*, pp 93–104. <https://doi.org/10.1109/ICPC.2019.00023>
33. Fontana FA, Mäntylä MV, Zanoni M, Marino A (2016) Comparing and experimenting machine learning techniques for code smell detection. *Empir Softw Eng* 21(3):1143–1191
34. Forman G, Scholz M, Rajaram S (2009) Feature shaping for linear SVM classifiers. In: *Proceedings of the 15th ACM SIGKDD international conference on knowledge discovery and data mining*, June 2009, pp 299–308
35. Ali PJM, Faraj RH (2014) Data normalization and standardization: a technical report. *Mach Learn Techn Rep* 1(1):1–6
36. Soto AJ, Cecchini RL, Vazquez GE, Ponzoni I (2008) A wrapper-based feature selection method for ADMET prediction using evolutionary computing. In: Marchiori E, Moore JH (eds) *Evolutionary computation, machine learning and data mining in bioinformatics*. *EvoBIO 2008. Lecture notes in computer science*, vol 4973. Springer, Berlin, Heidelberg. [https://doi.org/10.1007/978-3-540-78757-0\\_17](https://doi.org/10.1007/978-3-540-78757-0_17)