

A systematic analysis on machine learning classifiers with data pre-processing to detect anti-pattern from source code

Nazneen Akhter¹, Afrina Khatun¹, Md. Sazzadur Rahman², A. S. M. Sanwar Hosen³,
Mohammad Shahidul Islam²

¹Department of Computer Science and Engineering, Bangladesh University of Professionals, Dhaka, Bangladesh

²Institute of Information Technology, Jahangirnagar University, Dhaka, Bangladesh

³Department of Artificial Intelligence and Big Data, Woosong University, Daejeon, South Korea

Article Info

Received Jan 16, 2024

Revised Jul 10, 2024

Accepted Jul 26, 2024

Keywords:

Anti-pattern

Code quality

Code smell

Data pre-processing

Machine learning

ABSTRACT

Automatic detection of anti-patterns from source code can reduce software maintenance costs massively. Nowadays, machine learning approaches are very commonly used to identify anti-patterns. Hence, it is very crucial to choose a classifier that can be useful for detecting anti-patterns. This work aims to help practitioners to choose a suitable classifier to detect anti-patterns. In this paper, we highlight 16 classifiers in four different categories to detect anti-patterns. Furthermore, the performance of these classifiers is identified with the data pre-processing (DPP) to detect four commonly occurring anti-patterns from the three commonly used open-source Java projects' source code. The accuracy of Dagging classifiers is 98.4%. Kernel logistic regression (KLR) also performs well i.e., 97%. In the case of time complexity, naive Bayes (NB), decision trees (DT), support vector machines (SVM), library for support vector machines (LibSVM), logistic, and LightGBM (LB) have less time complexity to build a model in all the projects.

This is an open access article under the [CC BY-SA](#) license.



Corresponding Author:

A. S. M. Sanwar Hosen

Department of Artificial Intelligence and Big Data, Woosong University

Daejeon 34606, South Korea

Email: sanwar@wsu.ac.kr

1. INTRODUCTION

Software quality is very important as it defines the ease of maintainability, testability, readability, stability, speed, usability, size, cost, and security. It also ensures the smoothness, conciseness, and customer satisfaction of software. However, due to the heavy workload and time pressure, software quality issues are usually ignored. Some reports show that software evolution, understandability, usability, modularity, reusability, analyzability, and changeability are neglected during the development process of the software [1]. As a result, design flaws i.e., poor structure in the source code increase the development and maintenance cost. These design flaws are termed as anti-pattern or code smell. Palomba *et al.* [2] report that the maintenance cost of software is 2–100 times greater than the development cost. The term code smell (anti-pattern) is first introduced in [3]. Anti-patterns from a source code can be detected both by manual and automated processes. For large software projects, the manual detection system is a very time-consuming process as it mainly depends on proper documentation, source code structure, and developer's experience [1]. Hence, an automated and effective anti-pattern detection technique is essential. Although few studies are available in this field. Sometimes people may become confused about choosing machine learning (ML) classifiers and environmental

setup. Several researchers introduced three different support vector machines (SVM)-based approaches [4]–[6]. Maiga *et al.* did not adopt any data pre-processing (DPP) technique, while Akhter *et al.* adopted a DPP technique namely synthetic minority over-sampling technique (SMOTE) with SVM which performed better than synthetic minority under-sampling with random forest (SMURF). Five different ML classifiers: J48, random forest (RF), naive Bayes (NB), SVM, and just-in-time rule induction procedure (JRIP) to detect anti-pattern are used in [7]. They reported NB performed the best. They did a comparison among ML classifiers and detection of code and design smells (DECOR) (a method of code and design smell detection). They reported that in terms of F-measure and recall, the performance of DECOR was higher than NB but precision was lower. The author used 16 different classifiers and their boosting techniques (in total 32 classifiers) [8]. They reported that the best algorithm had approximately 97% of F-Score.

The aforementioned researchers hardly use any techniques to process the data. In comparison, a DPP technique can influence the performance of the classifier. As there are many classifiers most of the time researchers become puzzled to choose a classifier. Hence, an appropriate classifier selection is also a crucial study for anti-pattern detection. In this paper, a comprehensive study is conducted so that the practitioners can get a complete idea of choosing the classifier and their working process. A data-preprocessing technique is adopted to enhance the performance of the classifiers.

This is how the remainder of the paper is structured. The background research on anti-pattern detection is covered in section 2. In section 3, an explanation of the suggested methodology is provided along with an analysis of sixteen ML classifiers. Section 4 addresses the outcome and execution. Section 5 concludes this report by discussing future research.

2. LITERATURE REVIEW ON ANTI-PATTERN DETECTION

Inadequate design raises the system's development and maintenance costs by lowering performance and reusability. As a result, scholars developed various theories to address this issue. The three main types of methodologies used in anti-pattern detection research that are currently in use are rule-based, ML-based, and deep learning-based. Detailed discussion is provided in the following.

The main aim of the rule-based approach is to propose a set of rules to find anti-patterns. To do that, several researchers [9], [10] propose two different approaches to identify anti-patterns from the source code. Blob, spaghetti code, and functional decomposition by utilizing three heuristic search algorithms [9], [10]. Aras and Selcuk [11] present an automated approach using metrics and rules to detect anti-patterns in an object-oriented system. Three anti-patterns are used namely Blob, Swiss Army Knife, and Lava Flow. 36 Java classes are focused on conducting this experiment. However, the accuracy is not satisfactory. Due to issues with rule-based systems' accuracy, researchers introduced ML techniques. The primary issue with the rule-based technique is that each smell has its own set of rules, all of which are manually specified by the researcher.

A method for identifying design patterns utilizing ML-based techniques and software metrics has been put out by [12]. For small scale, code recall is 90% but for large scale code, it is 60%. Two SVM-based methods, SMURF and SVMDetect, are introduced by Maiga *et al.* [5], [6] to identify four anti-patterns: Swiss Army Knife, Blob, functional decomposition, and spaghetti code. The precision and recall for SMURF are 97% and 84.09% respectively (Blob Class). After computation, they report that SVMDetect can detect 143 anti-patterns whereas DETEX only detects 102. The recall and precision rates of SVMDetect are 84.09 and 97.09% respectively for (Blob class). In terms of anti-pattern identification, both methods perform similarly and can identify 143 instances of blobs within the system. Barbez *et al.* [13] present an ensemble method called smart aggregation of anti-patterns detectors (SMAD) to detect god class and feature envy. SMAD combined different approaches depending on their internal detection rule. Yin *et al.* [14] reported that accuracy is more than 85% for the classifier AB-J48-pruned, RF, and AdaBoost random forest (AB-RF) among 15 ML classifiers. Azadi *et al.* [15] propose a tool named WekaNose to investigate code smell in a system. They intend to characterize the instances that influence the detection of code smell or not. Cruz *et al.* [16] introduce a detection technique with seven different machine-learning algorithms for detecting four types of bad smells.

Kumar and Sureka [17] develop a predictive model to detect anti-patterns. They have mentioned that RF performed the best. Pritam *et al.* [18] examine the impact of code smell on the change tendency of a specific class in an item system which leads them to discover mistakes in expectation of progress inclination utilizing code smell. They use 4100 unique classes of 14 software systems after pre-processing. They have reported the sensitivity is 0.70 and specificity higher than 0.67 for multi-layer perceptron (MLP). Jesudoss *et al.* [19]

propose another code smell detection technique utilizing two algorithms such as SVM and RF. SVM acted as a classifier and the RF is used for predicting the range of data. From the above discussion, we hardly found any DPP adopted. As a result, the accuracy rate of the classifiers decreases. Most of the researchers used the same type of ML classifiers like SVM, RF, and NB. It is a matter of concern that other classifiers like ensemble or meta classifiers performance are missing to detect anti-pattern.

Kacem *et al.* [20] propose a hybrid learning approach to identify four code smells namely god class, data class, feature envy, and long method from 74 open-source systems. They have reported that the performance of god class was very good (god class precision 99.28%, recall 98.58%, F-measure 98.93%). Das *et al.* [21] come up with a technique named convolution neural networks (CNN) to detect code smells. They have achieved 97% accuracy for the Brain class and 95% for the Brain method. Barbez *et al.* [22] propose a method namely convolutional analysis of code metrics evolution (CAME) to detect anti-patterns. From the background study, it is found that DPP techniques are missing. ML classifiers can enhance their performance if the model gets pre-processed data. Recent research works are using similar kinds of ML classifiers (the most used classifiers are SVM, NB, and RF). Sometimes it becomes very tough to choose a classifier for the practitioner to work in this field.

3. ANTI-PATTERN DETECTION MODEL

In this paper, a constructive analysis of ML classifiers is presented to detect anti-patterns. Sixteen different types of classifiers have been adopted to conduct the study. Considering their nature, classifiers are divided into four subgroups: function classifiers, Bayes classifiers, tree classifiers, and meta classifiers [23]. Figure 1 represents the details of the proposed model. A detailed description of those is provided to understand the working principle of the classifiers deeply in the case of anti-pattern detection. The following describes the specifics.



Figure 1. The proposed methodology to detect anti-pattern from source code

3.1. Dataset and pre-processing

Three separate open-source Java projects—Azureus v2.3.0.6, Xerces v2.7.0, and ArgoUML v0.19.8 are employed in this study. ArgoUML is an application for creating unified modeling language (UML) diagrams. The ArgoUML dataset comprises 1,230 classes. Azureus with 1,449 classes uses the bit-torrent protocol to transmit data. The XML software library is called Xerces. There are 513 classes in the Xerces data file. This paper uses data sets supplied in [6]. To balance the data and improve the characteristics, we examined SMOTE as a DPP technique in this paper. This technique randomly chooses the minority class instance, and the k closest minority class neighbors are found. Afterward, a line segment in the feature space is formed by joining a and b, one of the k nearest neighbors (b), at random to generate the synthetic instance. The two selected examples, a and b, are used to create the synthetic instances by using the convex combination [24]. Consequently, new synthetic instances are generated that can be referred to as feature-boosting techniques.

3.2. Critical analysis of classifiers on anti-pattern detection

Functional classifiers mainly include the idea of neural networks and regression. The main procedure of a functional classifier is to assign data labels based on a function that is also known as the core function of the classifier. In this work, five functional classifiers are considered. The first classifier SVM performs the classification by calculating a boundary value known as a hyperplane that differentiates two data classes. SVM is broadly categorized into two groups; linear and non-linear SVM [25]. Non-linear SVM utilizes kernel tricks to classify data. The kernel function is used to transform lower dimensional input space into higher dimensional space. Kernel logistic regression (KLR) solves the classification problems by transforming the real input space into a high-dimensional feature space by utilizing kernel functions [26]. LibSVM is a complete package of SVM. This classifier can support various SVM formulations for classification [27]. In this project, support

vector classification (SVC) (two-class and multi-class) is used. Logistic regression predicts the odds ratio which is the event that needs to be predicted, for example, how likely a sample is an anti-pattern [28]. Multi-layer perceptron classifier (MLPCL) can train fully connected MLP networks with one hidden layer. For further processing, all attributes are standardized (it gives all features the same influence). In this project, conjugate gradient descent is used as it can reduce the time complexity.

The Bayes classifier provides a probabilistic model, that makes the most probable prediction for a new example. The Bayes theorem is used to calculate the conditional probability. Naive Bayes (NB) is a classifier that uses a probabilistic approach for the classification tasks. NB follows the following equation to calculate the posterior probability for each class [29]. The classes that have the highest posterior probability are the output of prediction.

Tree classifiers divide the training data into subsets by utilizing a sequence of conditional statements. Naive-Bayesian tree (NBTree) classifier combines a decision tree (DT) with a NB classifier [30]. The tree is constructed recursively to represent the attributes of the dataset. The nodes of the tree utilize the DT classifier. On the other hand, the leaves contain naive-Bayesian classifiers [31]. DT utilizes instances of data to create trees. When the tree construction is done, it prunes the trees to improve the model's performance. In this work, an advanced decision tree algorithm namely the C4.5 algorithm is used (called J48 in Weka) [32]. Decision stump (DS) is simply a DT that utilizes a single attribute for splitting [33]. It creates a single root tree of the given values using a top-down greedy approach. RF creates a forest of randomly generated DT. RF combines the output of the DT to form the final output. The number of trees defines the accuracy of the classifiers. For example, the more trees, the more accuracy is produced. In our implementation, 100 trees are created [34].

Meta classifiers can ensemble different classifiers to enhance the performance of the base classifiers. Logit boost (LB) is a meta classifier that is based on the AdaBoost classifier. Among the weighted sample, higher weights will be assigned to the unclassified data [35]. Real AdaBoost (RAB) is a boosting classifier for the binary classification task. Here, the weak learner algorithm provides class probability estimation [36]. Probability estimation:

$$P(x) = Pw(Y = 1|x)\epsilon[0, 1] \quad (1)$$

Here, w is the weight of the training data. In this study, the DS is used as the base classifier. The number of iterations is considered 10. It updates the weights on every iteration.

Bagging is an ensemble learner that randomly redistributes the original training dataset for individual base models [37]. Voting or averaging their prediction probabilities gives a final prediction. Diverse ensemble creation by oppositional re-labeling of artificial training examples (DECORATE) is designed to use additional artificially generated training data [37]. Additionally, it achieves equivalent performance on bigger training sets and gains higher accuracy than boosting on the small training set [38]. Dagging creates several disjoint, stratified folds of the data to predict the final output by averaging them [39]. Voting is another ensemble algorithm that creates two or more sub-models with the base classifiers to classify data. The final prediction is done by aggregating the sub-model predictions. In majority voting, the final prediction is made by combining the sub-models output. The class that gets the most number of votes will be the output of the classification [40].

4. RESULT ANALYSIS

The performance analysis of various ML classifiers is presented in this section. Five evaluation metrics are used: accuracy, F-measure, precision, recall, and receiver operating characteristic (ROC). An Intel Core i3 processor with 4 GB RAM and a 64-bit operating system is used for the experiment. WEKA is employed in the implementation of the classifier models. The details are described in the following.

4.1. Research questions and evaluation

Three research issues are addressed in this section to validate the anti-pattern detection technique. In RQ1, the significance of the DPP technique is briefly discussed. The performance of classifiers in terms of computing time is shown in RQ2. Each study question is thoroughly described and evaluated in the analysis that follows.

4.1.1. RQ1: How well do classifiers work with data pre-processing techniques to find anti-patterns?

In the project of ArgoUML, the Dagging classifier performs better in terms of accuracy than other classifiers. The accuracy of the Dagging is 98.4% when DS is used as a base classifier. The accuracy of DS as

an individual classifier is 91.37%. Dagging breaks the original training dataset into small chunks. With these small chunks of data, the base classifier can train itself well. So, it can classify data more accurately from the test dataset. For this dataset, the NB classifier performs poorly in terms of accuracy, 79.37%. This is because of the high dependency on the dataset attributes. The Dagging classifier is also used on the dataset using the NB as the base classifier. For that case, the accuracy of Dagging becomes 91.99%. Hence, it can be said that meta-classifiers like Dagging can enhance the performance of weak-performed classifiers. Table 1 shows the detailed result of the classifiers in terms of accuracy for ArgoUML, Azureus, and Xerces project. In the project of Azureus, data are non-linearly separable. The accuracy of KLR is 97%. This is because KLR can perform well with non-linearly separate data. KLR offers a natural estimate of the class probability. On the other side, the Dagging classifier performed poorly for this project. DS is used as the base classifier.

Table 1. Accuracy of the classifiers for ArgoUML, Azureus, and Xerces projects

Classifiers	Accuracy (ArgoUML) (%)	Accuracy (Azureus) (%)	Accuracy (Xerces) (%)
SVM	92.48	90.32	91.49
NB	79.37	76.51	73.71
RF	92.78	92.03	93.35
DT	90.3	91.15	87.76
LibSVM	96.55	90.37	95.82
Logistic	90.78	81.27	88.58
KLR	91.82	96.99	89.90
NBTree	88.03	85.31	86.60
MLPCL	91.86	93.93	90.285
RAB	92.31	92.61	93.08
Bagging	91.78	91.8	88.47
Dagging	98.37	68.69	96.04
LB	91.82	96.01	92.59
DECORATE	92.92	91.67	92.75
DS	91.37	87.26	89.02

Performance of tree classifiers: in this work, four tree classifiers have been used; DT, RF, NBTree, and DS. Among those, the RF classifier performs better than other classifiers for all three projects. The accuracy of RF for ArgoUML, Azureus, and Xerces are 92.8%, 92.03%, 93.4%, respectively. This is because the RF is an ensemble learner that finds the best features from the dataset randomly. The final prediction is done by combining the output of many decision trees. NBTree performs lower than other tree classifiers. The accuracy of NBTree for the three projects is 88.03%, 84.81%, and 86.61%, respectively. NBTree utilizes NB in the leaves of the tree. NB tree shows minimum performance for the dependent dataset which is the main reason behind performing low for NBTree. Performance of ensemble classifiers: for the project of ArgoUML, Dagging classifier performs the best whose accuracy is 98.48%. Figure 2 shows the detailed result of bagging, boosting, dagging, and DECORATE classifiers according to their accuracy. Here, DS is used as a base classifier.

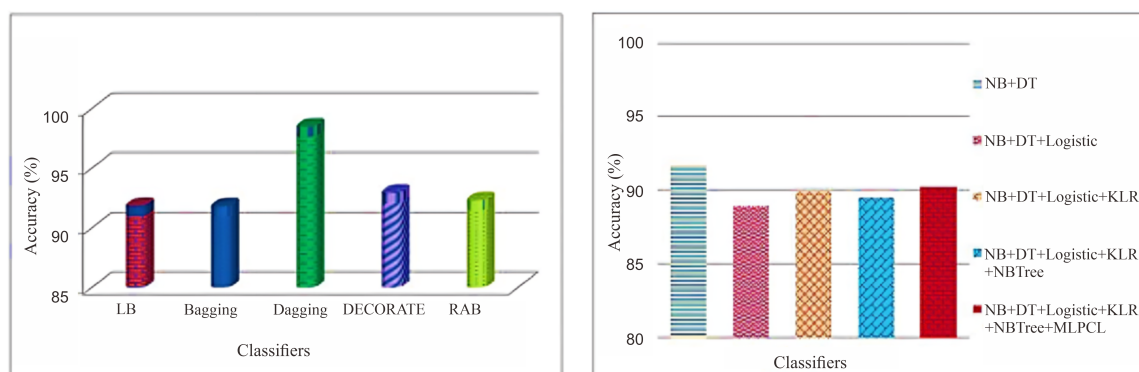


Figure 2. Performance of meta classifiers and voting classifier for ArgoUML project

Performance of voting classifier: in this work, two combination rules of voting are used; average voting and majority voting. The sub-models of voting classifiers are decided on the individual performance of the classifiers. For example, in the ArgoUML project, NB, DT, logistic, KLR, NBTree, and MLPCL classifier's performance is worse than other classifiers. So, they are used as the base classifier of voting for the ArgoUML project. In this work, some combinations of base classifiers are used to evaluate the result. First, the result of NB and DT is noted down. Then classifiers are added to check the performance of the Voting classifier one by one. Figure 2 reports the detailed result of the voting classifier according to their base classifiers. The accuracy of NB and DT are 79.37% and 90.3% respectively. When it is used as the sub-models of the voting classifier, the accuracy has been increased to 91.68% (both for average and majority voting). In the Xerces project, NB performs poorly than other classifiers in terms of accuracy. Other classifiers like DT, logistic, KLR, and bagging cannot provide satisfactory results either. So, the different combinations of classifiers are taken as a base classifier for the voting meta classifier to evaluate the result. When NB and DT are taken as a combination of base classifiers, the accuracy has been increased i.e., 90.61%. From this analysis, it can be said that voting classifiers can increase the performance (proper combination of base classifiers) than individual classifiers. Table 2 shows the result of the classifiers in terms of recall and accuracy.

Table 2. Recall and accuracy of classifiers with and without DPP for ArgoUML project

Classifiers	Recall (without DPP)	Recall (with DPP)	Accuracy (without DPP)	Accuracy (with DPP)
SVM	0.925	0.948	0.921	0.925
NB	0.739	0.833	0.739	0.793
RF	0.926	0.949	0.915	0.927
DT	0.903	0.935	0.901	0.903
LibSVM	0.954	0.964	0.953	0.965
Logistic	0.899	0.932	0.899	0.907
KLR	0.918	0.943	0.912	0.918
NBTree	0.931	0.941	0.875	0.88
MLP CL	0.916	0.943	0.916	0.918
RAB	0.919	0.945	0.918	0.923
Bagging	0.916	0.943	0.915	0.917
Dagging	0.981	0.983	0.974	0.983
LB	0.918	0.944	0.908	0.918
DECORATE	0.925	0.949	0.925	0.929
DS	0.913	0.942	0.913	0.913

In comparison with the DPP, we found most of the classifiers perform well. Dagging classifier performs better than other classifiers in terms of accuracy i.e., 97.4% whereas the accuracy of dagging is 98.3% with the DPP. Maiga *et al.* [5], [6] proposed two SVM-based works SMURF and SVMDetect. The recall of SMURF and SVMDetect is 84.09%. Whereas after using DPP the recall of SVM enhances to 94.8% in this work. Furthermore, from Table 3 we can see that, DPP is increasing the recall of the classifiers to detect anti-patterns.

Table 3. Comparison of proposed word with existing models in terms of recall

Authors	Model	Recall (%)
Maiga <i>et al.</i> [5]	SMURF	84.09
Maiga <i>et al.</i> [6]	SVMDetect	84.09
Proposed method	SVM with DPP	94.8

4.1.2. RQ2: How do the classifiers perform in terms of time?

In this work, sixteen different classifiers are taken into consideration. Most of the classifiers performed well in terms of accuracy and time. NB, DT, SVM, LibSVM, logistic, and LB have taken less time to build a model. They have taken 0.01 seconds to create the model. NBTree has taken the most time to build the model which is 0.68 seconds. The voting classifier also takes more time when NBTree is added with other classifiers. From the performance analysis of different classifiers on different projects, it can be said that NBTree performs poorly in terms of time than other classifiers. Figure 3 shows the detailed result of the time that is taken by the classifiers to build the model.

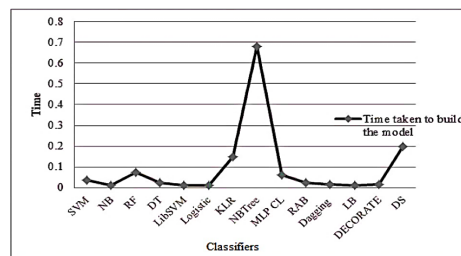


Figure 3. Time complexity to build the model for ArgoUML project

5. CONCLUSION

Anti-pattern is a significant nuisance in source code which violates the standard of design pattern. Software quality makes the software understandable, reusable, and cost-effective. Hence, anti-pattern detection has become a major concern. However, there are many difficulties in choosing appropriate classifiers to identify the design pattern. In this paper, a detailed discussion about 16 different well-known classifiers is done to detect anti-patterns. This analysis will help practitioners to choose suitable classifiers. In addition, to improve the performance of the classifiers, SMOTE DPP is used. Our classifiers performed better with DPP than the existing models without DPP. Dagging, KLR, and LibSVM perform better in terms of accuracy for respective projects. The accuracy of them is 98.4%, 97%, and 95.83% respectively. However, this work presents better accuracy than existing models. However, the data size used for this work is small. In the future, the dataset size can be increased by adding more samples. The data set can be applied to a deep learning model adopting an attention mechanism.

ACKNOWLEDGEMENTS

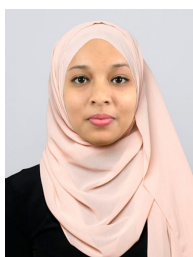
This work was supported by the Woosong University Academic Research Fund, 2024, South Korea.

REFERENCES

- [1] G. Rasool and Z. Arshad, "A review of code smell mining techniques," *Journal of Software: Evolution and Process*, vol. 27, no. 11, pp. 867–895, 2015, doi: 10.1002/smr.1737.
- [2] F. Palomba, G. Bavota, M. D. Penta, R. Oliveto, A. D. Lucia, and D. Poshyvanyk, "Detecting bad smells in source code using change history information," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2013, pp. 268–278, doi: 10.1109/ASE.2013.6693086.
- [3] M. Fowler, K. Beck, J. Brant, W. Opdyke, D. Roberts, and E. Gamma, *Refactoring: improving the design of existing code*. New York, United States of America: Pearson Education, 1999.
- [4] N. Akhter, S. Rahman, and K. A. Taher, "An anti-pattern detection technique using machine learning to improve code quality," in *2021 International Conference on Information and Communication Technology for Sustainable Development (ICT4SD)*, 2021, pp. 356–360, doi: 10.1109/ICT4SD50815.2021.9396937.
- [5] A. Maiga, N. Ali, N. Bhattacharya, A. Sabane, Y. -G. Gueheneuc, and E. Aimeur, "SMURF: A SVM-based incremental anti-pattern detection approach," in *2012 19th Working Conference on Reverse Engineering*, 2012, pp. 466–475, doi: 10.1109/WCRE.2012.56.
- [6] A. Maiga et al., "Support vector machines for anti-pattern detection," in *2012 27th IEEE/ACM International Conference on Automated Software Engineering*, 2012, pp. 278–281, doi: 10.1145/2351676.2351723.
- [7] F. Pecorelli, F. Palomba, D. D. Nucci, and A. D. Lucia, "Comparing heuristic and machine learning approaches for metric-based code smell detection," in *IEEE International Conference on Program Comprehension*, 2019, pp. 93–104, doi: 10.1109/ICPC.2019.00023.
- [8] F. A. 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, doi: 10.1007/s10664-015-9378-4.
- [9] M. Kessentini, H. Sahraoui, M. Boukadoum, and M. Wimmer, "Search-based design defects detection by example," in *Fundamental Approaches to Software Engineering*, Berlin, Heidelberg, 2011, pp. 401–415, doi: 10.1007/978-3-642-19811-3_28.
- [10] I. Polášek, S. Snopko, and I. Kapustik, "Automatic identification of the anti-patterns using the rule-based approach," in *2012 IEEE 10th Jubilee International Symposium on Intelligent Systems and Informatics*, 2012, pp. 283–286, doi: 10.1109/SISY.2012.6339530.
- [11] M. T. Aras and Y. E. Selcuk, "Metric and rule based automated detection of antipatterns in object-oriented software systems," in *2016 7th International Conference on Computer Science and Information Technology (CSIT)*, 2016, pp. 1–6, doi: 10.1109/CSIT.2016.7549470.
- [12] S. Uchiyama, H. Washizaki, Y. Fukazawa, and A. Kubo, "Design pattern detection using software metrics and machine learning," *First international workshop on model-driven software migration (MDSM 2011)*, vol. 708, pp. 38–47, 2011.
- [13] A. Barbez, F. Khomh, and Y.-G. Guéhéneuc, "A machine-learning based ensemble method for anti-patterns detection," *Journal of Systems and Software*, vol. 161, 2020, doi: 10.1016/j.jss.2019.110486.
- [14] Y. Yin, Q. Su, and L. Liu, "Software smell detection based on machine learning and its empirical study," in *Second Target Recognition and Artificial Intelligence Summit Forum*, 2020, doi: 10.1117/12.2550500.




- [15] U. Azadi, F. A. Fontana, and M. Zaroni, "Machine learning based code smell detection through WekaNose," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceedings*, 2018, pp. 288–289, doi: 10.1145/3183440.3194974.
- [16] D. Cruz, A. Santana, and E. Figueiredo, "Detecting bad smells with machine learning algorithms," in *Proceedings of the 3rd International Conference on Technical Debt*, 2020, pp. 31–40, doi: 10.1145/3387906.3388618.
- [17] L. Kumar and A. Sureka, "An empirical analysis on web service anti-pattern detection using a machine learning framework," in *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, 2018, pp. 2–11, doi: 10.1109/COMP-SAC.2018.00010.
- [18] N. Pritam et al., "Assessment of code smell for predicting class change proneness using machine learning," *IEEE Access*, vol. 7, pp. 37414–37425, 2019, doi: 10.1109/ACCESS.2019.2905133.
- [19] A. Jesudoss, S. Maneesha, and T. L. N. Durga, "Identification of code smell using machine learning," in *2019 International Conference on Intelligent Computing and Control Systems (ICCS)*, 2019, pp. 54–58, doi: 10.1109/ICCS45141.2019.9065317.
- [20] M. H. -Kacem and N. Bouassida, "A hybrid approach to detect code smells using deep learning," in *Proceedings of the 13th International Conference on Evaluation of Novel Approaches to Software Engineering*, 2018, pp. 137–146, doi: 10.5220/0006709801370146.
- [21] A. K. Das, S. Yadav, and S. Dhal, "Detecting code smells using deep learning," in *IEEE Region 10 Annual International Conference, Proceedings/TENCON*, 2019, pp. 2081–2086, doi: 10.1109/TENCON.2019.8929628.
- [22] A. Barbez, F. Khomh, and Y.-G. Gueheneuc, "Deep learning anti-patterns from code metrics history," in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2019, pp. 114–124, doi: 10.1109/ICSME.2019.00021.
- [23] R. K. Dash, "Selection of the best classifier from different datasets using WEKA," *International Journal of Engineering Research & Technology (IJERT)*, vol. 2, no. 3, pp. 1–7, 2013.
- [24] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "SMOTE: Synthetic minority over-sampling technique," *Journal of Artificial Intelligence Research*, vol. 16, pp. 321–357, 2002, doi: 10.1613/jair.953.
- [25] V. Jakkula, "Tutorial on support vector machine (SVM)," *School of EECS, Washington State University*, vol. 37, pp. 1–13, 2011.
- [26] W. Chen, X. Yan, Z. Zhao, H. Hong, D. T. Bui, and B. Pradhan, "Spatial prediction of landslide susceptibility using data mining-based kernel logistic regression, naive Bayes and RBFNetwork models for the Long County area (China)," *Bulletin of Engineering Geology and the Environment*, vol. 78, no. 1, pp. 247–266, 2019, doi: 10.1007/s10064-018-1256-z.
- [27] C. C. Chang and C. J. Lin, "LIBSVM: A library for support vector machines," *ACM Transactions on Intelligent Systems and Technology*, vol. 2, no. 3, 2011, doi: 10.1145/1961189.1961199.
- [28] S. L. Cessie and J. C. V. Houwelingen, "Ridge estimators in logistic regression," *Applied statistics*, vol. 41, no. 1, pp. 191–201, 1992.
- [29] I. Rish, "An empirical study of the naive bayes classifier," *IJCAI 2001 workshop on empirical methods in artificial intelligence*, vol. 3, no. 22, pp. 41–46, 2001.
- [30] R. Kohavi, "Scaling up the accuracy of naive-Bayes classifiers: A decision-tree hybrid," in *2nd International Conference on Knowledge Discovery and Data Mining, KDD 1996*, 1996, pp. 202–207.
- [31] T. M. Christian and M. Ayub, "Exploration of classification using NBTree for predicting students' performance," in *2014 International Conference on Data and Software Engineering (ICODSE)*, 2014, pp. 1–6, doi: 10.1109/ICODSE.2014.7062654.
- [32] J. R. Quinlan, *C4.5: Program for machine learning*. San Mateo, California: Morgan Kaufmann Publishers, 1993.
- [33] C. Sammut and G. I. Webb, "Decision stump," in *Encyclopedia of Machine Learning*, Boston, MA: Springer, 2011, pp. 262–263, doi: 10.1007/978-0-387-30164-8_202.
- [34] L. Breiman, "Random forests," in *Machine Learning*, Springer, 2001, pp. 5–32, doi: 10.1023/A:1010933404324.
- [35] B. Ziółko, S. Manandhar, R. C. Wilson, and M. Ziółko, "Logitboost WEKA classifier speech segmentation," in *2008 IEEE International Conference on Multimedia and Expo*, 2008, pp. 1297–1300, doi: 10.1109/ICME.2008.4607680.
- [36] J. Friedman, T. Hastie, and R. Tibshirani, "Additive logistic regression: a statistical view of boosting (with discussion and a rejoinder by the authors)," *The Annals of Statistics*, vol. 28, no. 2, pp. 337–407, 2000, doi: 10.1214/aos/1016218223.
- [37] L. Breiman, "Bagging predictors," *Machine Learning*, vol. 24, no. 2, pp. 123–140, 1996, doi: 10.1023/A:1018054314350.
- [38] P. Roy, R. Chakraborty, I. Chowdhuri, S. Malik, B. Das, and S. C. Pal, "Development of different machine learning ensemble classifier for gully erosion susceptibility in Gandheswari Watershed of West Bengal, India," in *Machine Learning for Intelligent Decision Science*, 2020, pp. 1–26, doi: 10.1007/978-981-15-3689-2_1.
- [39] J. Kittler, M. Hatef, R. P. W. Duin, and J. Matas, "On combining classifiers," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 20, no. 3, pp. 226–239, 1998, doi: 10.1109/34.667881.
- [40] S. Srivastava, "Weka: a tool for data preprocessing, classification, ensemble, clustering, and association rule mining," *International Journal of Computer Applications*, vol. 88, no. 10, pp. 26–29, 2014, doi: 10.5120/15389-3809.

BIOGRAPHIES OF AUTHORS






Nazneen Akhter completed her B.Sc. in Information and Communication Engineering and M.Sc. in Information and Communication Engineering from the Department of Information and Communication Technology of Bangladesh University of Professionals, besides several professional skills. She joined the Department of Information and Communication Technology as a lecturer in 2021. She is currently serving as a lecturer in the Department of Computer Science and Engineering at Bangladesh University of Professionals, Dhaka, Bangladesh. Her research areas of interest include software engineering, artificial intelligence and applications, and natural language processing. She can be contacted at email: nazneen.akhter@bup.edu.bd.






Afrina Khatun    obtained her B.Sc. in Software Engineering and M.Sc. in Software Engineering from the Institute of Information Technology, University of Dhaka. She joined the Department of Information and Communication Technology as a lecturer in 2021. Currently, she is working as an Assistant Professor at the Bangladesh University of Professionals in the Department of Computer Science and Engineering. Her research interest is the application of machine learning and deep learning in software engineering. She can be contacted at email: afrina.khatun@bup.edu.bd.






Dr. Md. Sazzadur Rahman    is working as a Professor at the Institute of Information Technology (IIT), Jahangirnagar University, Savar, Dhaka-1342, Bangladesh. He received his Ph.D. in Material Science (Surface Science) from Kyushu University, Japan as a MEXT scholarship grantee in 2015. Previously, he received his B.Sc. (Hons.) degree and M.Sc. degree in Applied Physics, Electronics, and Communication Engineering from the University of Dhaka, Bangladesh. He has about 14 years of experience in research and teaching in electronics, internet of things, machine learning, communication engineering, and material science. Now he is working as a co-investigator on the DIVERSASIA Erasmus+ project (project number 618615-EPP-1-2020-1-UKEPPKA2-CBHEJP). He can be contacted at email: sazzad@juniv.edu.



A. S. M. Sanwar Hosen    received the M.S. and Ph.D. degrees in computer science and engineering from Jeonbuk National University, Jeonju, South Korea. He is currently an Assistant Professor at the Department of Artificial Intelligence and Big Data, Woosong University, Daejeon, South Korea. He has published several articles in journals and international conferences. His current research interests include wireless sensor networks, the internet of things, fog-cloud computing, cyber security, artificial intelligence, and blockchain technology. He has been an expert reviewer for IEEE Transactions, Elsevier, Springer, and MDPI journals and magazines. He has also been invited to serve as the Technical Program Committee Member for several reputed international conferences, such as IEEE ACM. He can be contacted at email: sanwar@wsu.ac.kr or sanwar@jbnu.ac.kr.



Mohammad Shahidul Islam    received his Ph.D. in Computer Science and Information Systems from the National Institute of Development Administration (NIDA), Bangkok, Thailand, B.Tech. in Computer Science and Technology from the Indian Institute of Technology-Roorkee (IITR), Uttar Pradesh, India in 2002, M.Sc. in Mobile Computing and Communication from University of Greenwich, London, U.K in 2008. His research interests include artificial intelligence, robotics, machine learning, pattern recognition, and renewable energy. He can be contacted at email: suva.93@grads.nida.ac.th or suva93@gmail.com.