Proceedings of the 5th International Conference on Inventive Research in Computing Applications (ICIRCA 2023)
IEEE Xplore Part Number: CFP23N67-ART; ISBN: 979-8-3503-2142-5

Code Smell Detection using Hybrid Machine Learning Algorithms

Mahalakshmi D[1]
Assistant professor
Dept. of Information Technology
Panimalar Engineering College
mahalakshmi1607cs@gmail.com

Prabakaran Kasinathan[2]
Assistant Professor, Dept. of CSE
School of Computing, Veltech Rangarajan Dr. Sagunthula R & D Institute of Science and Technology,
kptpraba@gmail.com

D.Elangovan[3]
Associate Professor, Dept. of CSE,
Panimalar Engineering College,
elangovan.sdurai@gmail.com

C.Rohith Bhat[4]
Professor,Institute of computer science and engineering,
Saveetha School of engineering, (SIMATS), Chennai
rohithbhat2000@gmail.com

M Balamurugan[5]
Assistant Professor, Dept. of CSE
Sri Sairam Engineering College
Affiliated to Anna University
balamurugan.cse@sairam.edu.in

S.Sivakumar[6]
Assistant Professor, Dept. of Computer Applications
Lakshmi Bangaru Arts and Science College, melmaruvathur
sivaskumarmca@gmail.com

2023 5th International Conference on Inventive Research in Computing Applications (ICIRCA) | 979-8-3503-2142-5/23/$31.00 ©2023 IEEE | DOI: 10.1109/ICIRCA57980.2023.10220911

***Abstract*— Code smells frequently leads to the discovery of decreased code quality, drains on application resources, or even critical security vulnerabilities embedded within the application's code. While code smells may not always indicate a particularly serious problem, it do often lead to the discovery of these issues. Software's structural characteristics lead to a design issue that makes it challenging to manage and maintain code refactoring. The goal of the current research is to create methods for identifying code smells. The machine learning algorithm is a reliable method for individualized smell detection, but there aren't many studies on how well it works for different developers. In this proposed work used two different deep learning algorithms and five different machine learning ensembles to detect suspicious code. Investigation of the Data class, God class, Feature-envy, and Long-method datasets revealed that each contained various levels of code smells. Although there is room for improvement, the outcomes of prior publications' applications of machine learning and stacking ensemble learning methods to this dataset were satisfactory. A class balancing method (SMOTE) was implemented to address the problem of class imbalance within the datasets. While the Feature-envy dataset with the selected dozen metrics produced the lowest accuracy (91.45%) for the Max voting method, the Long-method dataset with the various chosen metrics produced the highest accuracy (100%) for all five methods.**

***Index Terms*—** Code smell, Deep Learning, Code generation, ensemble Learning, feature selection and Machine Learning

I. INTRODUCTION

Code Smells are not the program's mistakes. Even if your program has code smells, it might still work. It just shows where the design is weak and could make bugs and program failures more likely in the future. Fowler analyses 22 code smells and links each one to a specific refactoring transformation that can be used to fix the underlying problem and make the code more readable and maintainable. He uses a loose definition of "code smells" and suggests that human intuition alone can help determine if reorganization is long overdue. A method has Feature Envy when it uses the features of a class different from that where it is declared, which may indicate that the method is misplaced, or that some pattern as Visitor is being applied [1]. A code smell known as feature envy occurs when one class "envies" another class's capabilities. Because the class in question has such a strong desire to belong to the other class, it makes excessive use of the methods that class employs. The creation of code frequently requires teamwork. In order to prevent any one class from becoming excessively large and complicated, multiple classes are created. Long methods have too many lines. Code over 25 lines should be questioned. Refactoring techniques like sub-expressions and complex codes into new functions can make a function or method small and easy to read without changing its functionality. Long Parameter List: Complex functions have more parameters. Functions should have no more than 3–4 parameters. To fix the code smell, check the parameter values and pass them as functions if they are the output of another function. Oversampling minorities can help balance datasets. Duplicating minority class examples doesn't improve the model, but it's the easiest method. Synthesize new examples from existing ones. The Synthetic Minority Oversampling Technique (SMOTE) augments minority data. While some code smells specify actual problems in the code (such as long argument lists making methods difficult to implore), others are just possible indications of a problem. As pointed out by [2], Fowler's concepts are far too ad hoc to be used in a machine learning-based olfactory recognition system. Scent recognition is greatly influenced by the ambiguity around olfactory terminology. In this section, identified a few code smells are discussed. Code smells are the attributes of a software system which signifies a design and code issue also makes the

softwaredifficult to progress and pressure. It is the manifestation ofthe execution choices which causes serious issue for additional expansion, progress and advancement of software system. In order to abolish code smell, to refine the supportability and also the software progress, it's important to focus on refactoring steps to intensify the inner caliber of the software [3]. Many additional refactorings may be used to accomplish the mechanics of a refactoring, creating a nested 'chain' of refactorings as dependencies. In this study, a methodology that might serve as a rough estimate of the time and energy needed to eliminate code odours is provided. This view is grounded in a quantitative examination of the interdependencies between Kerievsky's and Fowler's collections of code smells. To acquire the desired dependency information, an illustrative tool was developed. According to the findings, certain "code smells" need more work to fix than others do, therefore developers shouldn't get rid of the former just to get rid of the latter. Kerievsky's and Fowler's scents were found to be fundamentally different from one another [4]. Because of this implementation of detecting proficiency, the tools can improve the characteristics of the software system. In this work, It focus on code smells and the automated techniques that have been developed to help find them. Fowler, who classified odours into 22 categories, offered this idea. Over time, other authors recognised many more odours, and more sophisticated ones were uncovered [5].

Researchers have created a suite of automatic code scent detection techniques due to the high cost, low efficiency, and high mistake rate of manual code analysis for code smell detection. Code smells in big projects may be detected by a variety of technologies that employ various object-oriented source code metrics [6]. There are, nevertheless, still flaws in their design. Blob, Long Parameter List (LPL), Long Method (LM), and Feature Envy (FE) were the four types of odours that Amorim and his colleagues utilised the tools PMD, Check Style, JDeodorant, and inFusion to identify in a Gantt Project. Take note of how little consensus there is among these many instruments. The tools' divergent labelling makes it nearly impossible to ensure their accuracy. To solve this problemHowever, automatic tools can be useful in reducing the burden of searching for code smells in big code bases. All code smells should be eliminated, although this is not always the case and depends on the system. It is best to get rid of them as soon as possible when they express a desire to be eliminated. In order to get rid of bad odours in the code, In this work must first find the source of the problem [8]. Despite the ambiguity and openness of the idea of code smell, evaluating the efficacy of code smell detection technologies is a difficult endeavour in and of itself. When analysing the same system, several methods may come up with varying findings for a variety of reasons: One is that the scent definitions are vague, allowing for differing interpretations by the developers of various instruments. Words like "many," "few," "enough," "huge," and "intimate" in the procedure employed to comprehend the intensity of odours are completely imprecise. Other times, the criteria are too broad and need to be narrowed in order to make the detection tools more effective. Another issue is that various instruments employ various detecting methods [9]. They often rely on the calculation of a specific set of combined metrics, on conventional object-oriented metrics, or on metrics generated ad hoc for the purpose of odour detection.

When comparing measures that are otherwise similar but are measured in various ways, one major difference is the threshold values chosen. These values are defined by software engineers and programmers, thus it's important to take into consideration their knowledge and experience as well as external elements like the domain and scale of the system. The quantity of odours that can be identified is obviously greatly affected by the threshold. Generally ifwe validate the outcome of approaches that are existing it will be scarce, which can be done only on small system that is applicable to few smells. The analysis of the detected results is complex because of the problems relevant to the manual validation. The results which are manually validated may or may not be completely correct, because they too have small criteria [10]. As a matter of fact, the value of some metrics can evolve over time, moving from human assessment by a programmer to automatic computation by the tool. Every coder has their own unique way of manually computing, though. Despite the fact that there is a trend towards increasing congruence between the results of manual and computerised code smell detection using a simple measure.

II. LITERATURE SURVEY

Code smell detection is an area where many different methods have been introduced. According to the MLT established [11], code smells may be categorised based on their degree of severity. This technique can help programmers arrange classes or methods in a certain hierarchy. A multinomial classification and regression technique is used to rate the severity of code smells. The bug severity reports prediction for proprietary datasets was proposed in [12]. The PROMISE data warehouse served as the source for the NASA project dataset (PITS). With the use of ensemble methods and two-dimensional reduction strategies (Chi-square and information gain), they were able to get higher precision. They found that the bagging method outperformed the other ensemble techniques. In order to determine the importance of a reported problem in closed-source software presented eight MLTs [13]. These issues are to proprietary software developed by INTIX, a Jordanian firm with headquarters in Amman. In order to compile their data, they used the JIRA issue tracker. After comparing the performance of several MLTs, they settled on the decision tree method. Ensemble techniques combining supervised and unsupervised classification for bug severity reports in closed source datasets were reported by [14]. Selecting the right characteristics from the severity dataset was done with the use of information gain and Chi-square FSA. Results for Pits C ranged from 79.85% to 89.80% accurate. Code smell identification using MLTs is the primary focus of the

aforementioned publications. Most of the prior research that has been applied to the MLT has looked at a small number of systems [15] used six MLTs, a tuning optimization approach based on grid search, wrapper-based and Chi-square FSA to pick the right features from each dataset, and they got 100% accuracy with the logistic regression model on the LM dataset, but the accuracy of the other datasets, i.e. DC, GC, and FE, was not good.

III. PROPOSED WORK

This research makes use of four different code smell datasets (God class, Data-class, Feature-envy, and Long-method) provided by Fontana et al. [1] in order to construct the framework for the detection of code smells. In this research work present a method that makes use of developers' context in order to prioritise code smells. To help prioritise code odours, it establish a context relevance index (CRI) based on automated effect analysis. They provide actual research on the features of this method and how they relate to ranking quality. By presenting a well crafted experiment, we demonstrate that our method may effectively prioritise the code smells that are most highly regarded by industry experts.

This paper detects code smells using code smell detectors.. It also helps the user not only to find the number of loops keywords classes and methods also it highlights them. In this paper make use of a hybrid algorithm that combines SVM and random forest. The tools used here are PMD ,check style, JDeodorant and infusion. Once the code snippet is put into the process, the software metrices led them to the code smell detector. The code smell detector detects the code smell and send them to the classifiers .the classifiers segregates the different types of the code smells and send them for refactoring .refactoring is nothing but modifying the code snippet which contains the code smell so that the code snippet can be improved without any change in external behavior. After refactoring process, the code enters into cross validator. Cross validator undergoes cross-validation process. Cross-validation is a technique which is used for the assessment of evaluating our model and to detect overfitting (failing to generate the pattern). Atlast statistical report is generated in theform of graph.

PMD: PMD is a code analyzer designed to spot typical programming errors including those caused by unused variables, empty catch blocks, and the creation of objects that aren't needed.

Checkstyle: Checkstyle offers the ability to check many different facets of your source code, including class design flaws, method design problems, and code layout and formatting errors.

jDeodorant: The jDeodorant Eclipse plugin is a free and open source tool that can sniff out four common Java programming odours: God Classes, God Methods, Feature Envy, and Type Checking.

Infusion: infusion is a commercial standalone tool which is used to detect 22code smells. Here it is used to diagonise the bad smells found in our code snippet.

The code snippets from the software are sent for Developer Evaluation and Software metrices. The Developer evaluates the given code snippets and the software metrices checks the code and make them to sent for code smell detector. The code smell detector is used to find the false unused and dead codes . Then it is sent to the classifier algorithm. The classifier classifies thecode snippets depending upon the nature of the code and the classifier consist following codes namely J48 algorithm, Random forest algorithm, Decision tree algorithm and SVM algorithm. After analysis a graph static record is drawn for further reference.

There are three modules of work:

1. Analysis of number of code smells
2. Tracking and Threats to validity
3. Code smell Identification using Hybrid algorithm

Analysis of number of code smells:

When it comes to software, the term "code smell" refers to any symptom in the source code that may suggest a deeper problem, preventing software maintenance and evolution. Code smells are difficult for developers to identify, and their nebulous description has resulted in the proliferation of many detection methods and tools. Code smells are difficult to define since they depend on the language, the developer, and the development process. The purpose is to find out if there is an upward trend in the occurrence of code smells. They looked at how many times each code smell was mentioned in the Mobile Media and Health Watcher reference lists.

Tracking and Threats to validity:

Static analysis refers to the practise of inspecting and reporting on a program's source code for a variety of characteristics, but it also refers to the philosophical concept of viewing source code as data. As application developers, find this quite perplexing because it often associate source code with things like procedures and algorithms. However, its force lies in its depths. Ascertaining the development of code odours over time in relation to OS releases. Each code odour is tracked through its Code Smell Progression. It began by picking the examples from the bibliography. Classes and functions that gave off a bad odour in the code. Afterward, it followed their evolution throughout all of the different iterations.

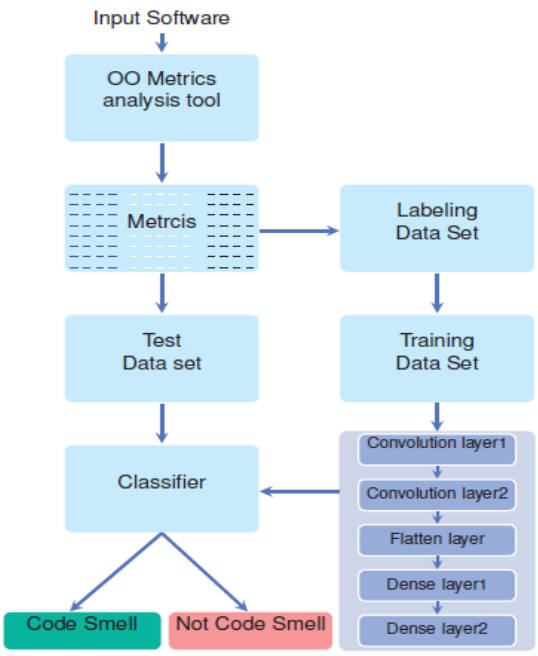

Fig. 1. Proposed Model

Code smell Identification using Ensemble Method

In this study, the research employ ensemble techniques to develop a model for identifying these "code smells." Fig. 1 depicts the stages of this framework. They started by picking the most popular code smell datasets [16]. After that, they used min-max normalization to scale features. Then, they used a method called SMOTE to equalize the classes. The best features were then extracted from the datasets using Chi-square FSA. The data was processed using ensemble and deep learning techniques. Our ten-fold cross-validation procedure was used to enhance the efficiency of our ensemble and deep learning approaches. After all was said and done, they calculated several metrics of success.

AdaBoost: It was Yoav Freund and Robert Schapire who first uncovered AdaBoost. When it comes to binary classification, AdaBoost was the first successful boosting technique. The Boosting technique takes numerous "weak classifiers" and turns them into one "strong classifier" [17].

Bagging: Bagging, or Bootstrap aggregation, is a form of ensemble MLT that simplifies the process of enhancing the MLT's performance and accuracy. The agreement's bias and variance are optimised to decrease the prediction model's inaccuracy. Bagging is a technique used in regression and classification models to avoid overfitting the data [18].

The Max Voting technique is an MLT that generates results (class) based on the class with the highest probability by using a collection of ensemble methods. It just adds up the results of all the classifiers that were fed into a voting classifier and makes a prediction based on which class received the most votes. Instead of developing separate single models and evaluating their effectiveness [19], it is more common practise to design a single model that teaches on several models and estimates outputs based on the aggregate number of votes for each output class.

The gradient boosting (GB) approach is the most efficient ensemble MLT. The most typical types of error made by MLTs are bias error and variance error. For example, the GB algorithm is a boosting technique that might be applied to lessen the algorithm's bias error. The GB technique is used not just with linear models' constant target variables, as in regression, but also with classifiers' categorical target variables. When doing a regression, the cost function is the mean square error (MSE), but when performing a classification, the cost function is the log loss [20].

XGBoost is the extreme gradient boosting method. It's an enhanced version of the tree-based MLA in terms of readability and speed. The DMLC (Distributed Machine Learning Community) is primarily responsible for maintaining and updating XGBoost, which was developed by Tianqi Chen. It has become well-liked since it effectively produces useful outcomes in organised and tabular information.

Convolutional Neural Networks (CNN) that learn to "convolve" are one of the most popular and well-known deep learning techniques. CNN's main advantage over its competitors is its ability to detect crucial parts automatically, which has contributed to its widespread adoption [21].

IV. RESULTS AND DISCUSSIONS

Accuracy is a metric that evaluates how well PPV and sensitivity correlate with one another. The chart shows the proportion of correctly classified positive and negative occurrences. For the purpose of precision calculation, they employed formula (1). The percentage of correct answers may be found by dividing the sum of all correct and incorrect answers by the sum of all correct, incorrect, FP, and FN answers.

$$\text{Accuracy} = \frac{TP+TN}{TP+TN+FP+FN} \qquad (1)$$

In this paper tested the performance of five ensembles and one deep learning method. The results of all experiments for each ensemble and deep learning technique and comparisons are displayed in Table 1 in terms of accuracy.

Table 1 Outcomes of Ensemble algorithms with and without applied SMOTE

S. No.	Model Name	Accuracy
1	AdaBoost	99.15
2	Bagging	99.18
3	The Max Voting	99.14
4	Gradient Boosting	100

5	XGBoost	100
6	CNN	99.29

Some other academics have also used the same code smell datasets in their own studies. They implemented machine learning and stack ensemble learning methods. Evaluated results were compared to those of similar research in this section. Table 2 displays these data and fig. 2. displays the graph of comparison.

Table 2 Result comparison of our approach with other correlated works.

S. No.	Model Name	Accuracy
1	B-J48 Pruned	99.02
2	RF and J48	83
3	RF	99.70
4	Stack-LR	98.92
5	Proposed Model	100

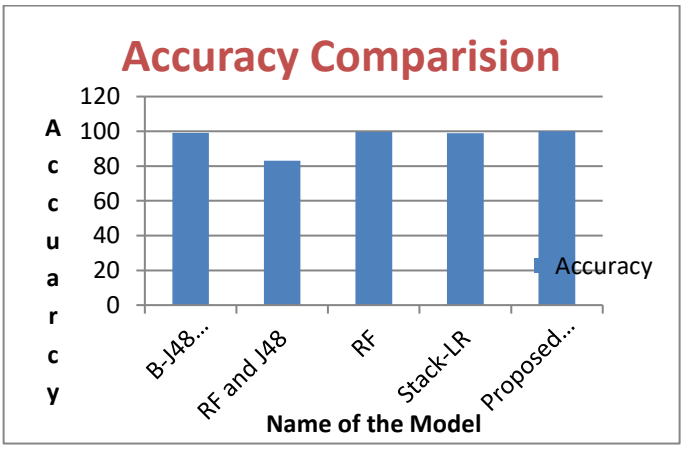

Fig. 2. Accuracy Comparison Graph

In order to detect the code smell the file name should beentered and the directory is chosen then the process button should be clicked then it get executes and process the output. The output displays the file count, class count, method count, constructor count, variable count, total line count with andwithout comments and total size. Now the code smell is detected. The encountered code smell type is chosen in bloatedcode detector then it will display the class name file name and parameter details.

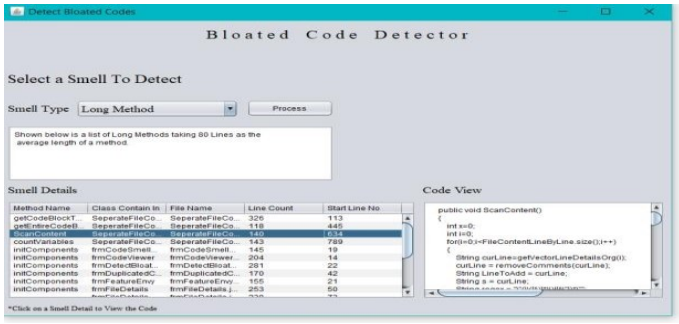

Fig.3. Bloated Code Detector

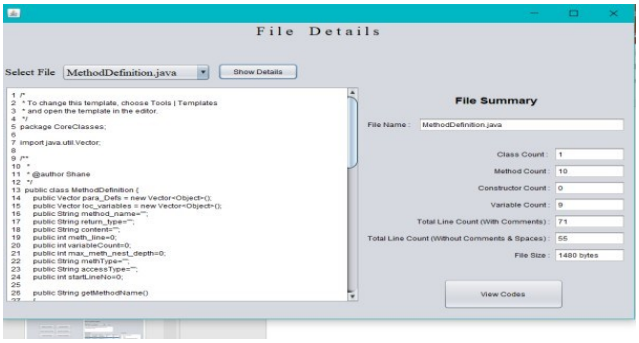

Fig. 4. Report of Code Smell

The encountered code smell type is chosen in bloated code detector then it will display the number of classes within the particular smell, number of methods along with their names, number of attributes, the starting line number of the particular code smell and it also helps to view the code snippet .In addition to that the lazy class detector produces an average linecount, average method count, average variable count and italso displays the code. This is known as Report generation

V. CONCLUSIONS AND FUTURE WORKS

In this study, the author proposes a hybrid approach to the detection of four code smells. This approach makes use of both unsupervised and supervised algorithms. In this work presented the results of a study that evaluated the accuracy of ensemble-algorithms on the detection of code smells for different developers. To identify these "code smells," this study recommended a combination of ensemble and deep learning techniques. Five ensemble MLTs and a deep learning CNN are used to identify the code odours. This study has a dual purpose: The first fold used ensemble methods to find suspicious code, while the second fold compared and computed the performance of these ensemble MLTs based on their correctness. Because machine learning shows promise as a bespoke method of code smell detection, this study is crucial. However, there appears to be no research assessing how well ML-algorithms can adapt their detection to accommodate developers' varying intuitions regarding code odours. Here, author used a hybrid

classification strategy to sniff out two code smells at the method level. Previous research only looked for one specific kind of "code smell," while the suggested study looked for two other types of "code smell," regardless of whether they were present in the same technique or not. Two datasets built using single-type detectors were considered for this investigation.

REFERENCES

1. F. A. Fontana, M. V. Mäntylä, M. Zanoni, and A. Marino, ''Comparing and experimenting machine learning techniques for code smell detection,'' Empirical Softw. Eng., vol. 21, no. 3, pp. 1143–1191, Jun. 2016.
2. Puneeth RP, Parthasarathy G. A Survey on Security and Interoperability of Electronic Health Records Sharing Using Blockchain Technology. Acta Informatica Pragensia. 2023,12(1).
3. D. Sahin, M. Kessentini, S. Bechikh, and K. Ded, ''Code-smells detection as a bi-level problem,'' ACM Trans. Softw. Eng. Methodol., vol. 24, no. 1, p. 6, 2014
4. M. N. Pushpalatha and M. Mrunalini, ''Predicting the severity of open source bug reports using unsupervised and supervised techniques,'' Int. J. Open Source Softw. Processes, vol. 10, no. 1, pp. 1–15, Jan. 2019.
5. M. Y. Mhawish and M. Gupta, ''Generating code-smell prediction rules using decision tree algorithm and software metrics,'' Int. J. Comput. Sci. Eng., vol. 7, no. 5, pp. 41–48, May 2019.
6. G. Himanshu, T. G. Kulkarni, L. Kumar, L. B. M. Neti, and A. Krishna, ''An empirical study on predictability of software code smell using deep learning models,'' in Proc. Int. Conf. Adv. Inf. Netw. Appl., 2021, pp. 120–132, doi: 10.1007/978-3-030-75075-6_10.
7. Navaneethakrishnan, M., Vairamuthu, S., Parthasarathy, G. and Cristin, R., 2021. Atom search-Jaya-based deep recurrent neural network for liver cancer detection. IET Image Processing, 15(2), pp.337-349.
8. Dhanalakshmi, R., Bhavani, N.P.G., Raju, S.S., Shaker Reddy, P.C., Marvaluru, D., Singh, D.P. and Batu, A., 2022. Onboard Pointing Error Detection and Estimation of Observation Satellite Data Using Extended Kalman Filter. *Computational Intelligence and Neuroscience*, *2022*.
9. Majji, Ramachandro, et al. "Social bat optimisation dependent deep stacked auto-encoder for skin cancer detection." IET Image Processing 14.16 (2020): 4122-4131.
10. Chandana, C. and Parthasarathy, G., 2020, December. A comprehensive survey of classification algorithms for formulating crop yield prediction using data mining techniques. In 2020 IEEE International Conference (TEMSMET) (pp. 1-5). IEEE.
11. Fontana, F.A. and Zanoni, M., 2017. Code smell severity classification using machine learning techniques. *Knowledge-Based Systems*, *128*, pp.43-58.
12. Pushpalatha, M.N. and Mrunalini, M., 2019. Predicting the severity of closed source bug reports using ensemble methods. In *Smart Intelligent Computing and Applications: Proceedings of the 2nd Inter. Conf. on SCI 2018, Volume 2* (pp. 589-597). Springer Singapore.
13. Baarah, A., Aloqaily, A., Salah, Z., Zamzeer, M. and Sallam, M., 2019. Machine learning approaches for predicting the severity level of software bug reports in closed source projects. *International Journal of Advanced Computer Science and Applications*, *10*(8).
14. Pushpalatha, M.N. and Mrunalini, M., 2021. Predicting the severity of open source bug reports using unsupervised and supervised techniques. In *Research Anthology on Usage and Development of Open Source Software* (pp. 676-692). IGI Global.
15. Dewangan, S., Rao, R.S., Mishra, A. and Gupta, M., 2021. A novel approach for code smell detection: an empirical study. *IEEE Access*, *9*, pp.162869-162883.
16. Parthasarathy, G., et al. "Improvised approach for real time patient health monitoring system using IoT." Intelligent Systems and Computer Technology 37 (2020): 78..
17. Ganaie, M.A., Hu, M., Malik, A.K., Tanveer, M. and Suganthan, P.N., 2022. Ensemble deep learning: A review. *Engineering Applications of Artificial Intelligence*, *115*, p.105151.
18. G. Parthasarathy and DC.Tomar, "Trends in citation analysis", Intelligent Computing, Communication and Devices, Springer, New Delhi, pp.813-821, 2015.
19. Teng, L., 2022. Gradient boosting-based numerical methods for high-dimensional backward stochastic differential equations. *Applied Mathematics and Computation*, *426*, p.127119.
20. Alzubaidi, L., Zhang, J., Humaidi, A.J., Al-Dujaili, A., Duan, Y., Al-Shamma, O., Santamaría, J., Fadhel, M.A., Al-Amidie, M. and Farhan, L., 2021. Review of deep learning: Concepts, CNN architectures, challenges, applications, future directions. *Journal of big Data*, *8*, pp.1-74.
21. G P, Lakshmanan L, Ramanathan L. Using Citation Context to Improve the Retrieval of Research Article from Cancer Research Journals. Asian Pac J Cancer Prev. 2019 Mar 26;20(3):951-960.