Journal of Information Systems Engineering and Management
2025, 10(3)
e-ISSN: 2468-4376
https://www.jisem-journal.com/

Research Article

Enhancing Software Quality Prediction through Source Code Analysis with the Firefly Algorithm

[1]M. Muthalagu,[2]Dr. R. Rathinasabapathy
[1.]Research scholar, Deparment of Computer Applications ,Madurai Kamaraj University, Madurai,Tamilnadu, India. E-mail:muthalagucs76@gmail.com.
[2.]Associate Professor, Department of Computer Applications, Madurai Kamaraj University, Madurai, Tamilnadu, India. E-mail:saba.dca@mkuniversity.ac.in

ARTICLE INFO **ABSTRACT**

Received: 08 Nov 2024

Revised: 29 Dec 2024

Accepted: 23 Jan 2025

For software systems to operate reliably and effectively, software quality assurance is essential. Conventional techniques for predicting software quality frequently rely on manual code inspection and testing, which can be laborious and prone to errors. This study suggests a unique method for predicting software quality assurance through deep learning and data mining analysis of source code. Developing an automated system that forecasts software quality using source code analysis is the primary goal of the project. In order to identify significant patterns and characteristics from the source code and capture both structural and semantic information, the suggested system makes use of data mining techniques. A type of deep learning model employed to understand the intricate connections between software and the extracted features is the convolutional neural network (CNN). A large collection of source code samples and related quality metrics will be gathered in order to achieve this goal. The source code samples will be used to extract several code metrics, including code complexity, code duplication, and code smells. The data mining and deep learning models will use these metrics as input features. Pre-processing will be applied to the gathered dataset to address any noise or inconsistent data. The most pertinent and instructive elements for software quality prediction will be found using feature selection and dimensionality reduction approaches. Using the quality metrics and extracted features, deep learning algorithms will be developed and optimized using the training set. The models will undergo optimization processes, including hyper parameter tuning and regularization, to achieve optimal performance and generalization capabilities. The trained models will be evaluated using the validation set, fine- tuning them if necessary.

Keywords: Code smell, Data mining, Deep learning, Software quality, Source code analysis.

1. Introduction

When assessing the reliability and efficiency of computer programs, software quality assurance plays a vital role. The notion of "code smells"
[1] suggests that there is an opportunity to improve the source code of software systems throughout both development and maintenance phases. Effective code refactoring hinges on the precise and dependable identification of these code smells within the target code. Many automated techniques for detecting code smells utilize analytical and learning-based methods [2], which necessitate a substantial dataset with annotated examples of each smell. Therefore, the quality of the dataset is crucial for the creation and assessment of code smell detection tools [3]. Some code smell detection algorithms have been found to produce inaccurate results [3, 4], mainly due to issues with how their datasets were constructed and applied. Understanding the current leading datasets, including their advantages, limitations, and their use in detecting code smells, is essential. Various methods have been developed in this field, and Fontana et al. [5] have proposed a multi-level taxonomy (MLT) for evaluating code smells.

Programmers can organize the hierarchy of classes and methods with the use of this methodology. The intensity of code smells is assessed through a multinomial classification and regression approach. M.N. Pushpalatha et al. [6]

developed a technique to predict the severity of defect reports in proprietary datasets. They utilized the NASA project dataset (PITS), which was obtained from the PROMISE data repository. They used two- dimensional reduction tactics with ensemble methods to increase precision. Their results show that the bagging method works better than other ensemble methods [7]. The eight MLTs provided by Aladdin et al.
[8] can be used by closed source software projects to assess how serious a software bug report is. These problems relate to in-house software created by the Jordanian company INTIX in Amman. Their data set was built on top of the JIRA issue tracker. They discovered that the decision tree algorithm yielded the best results when integrated with other machine learning methods. Pushpalatha et al. [9] proposed ensemble methods that leverage both supervised and unsupervised learning to analyze bug severity in closed-source datasets. They employed information gain and Chi-square feature selection methods to pinpoint crucial features within the severity dataset, attaining accuracy rates between 79.85% and 89.80% for PITS C. Most of the referenced studies focus on code smell detection using multi-level taxonomy (MLT).

However, many earlier MLT-based studies have analyzed only a limited number of systems. Certain scholars have utilized a range of different feature selection algorithms (FSAs). Dewangan et al. [9] applied a tuning optimization method that merged grid search, wrapper-based approaches, and Chi-square feature selection analysis to pinpoint the key features for each dataset. They created a logistic regression model that attained perfect accuracy on the LM dataset but produced only average results on the DC, GC, and FE datasets.

Gogullothu et al. [20] examined datasets related to multi-label code smells. To assess the reproducibility of studies on code smell detection, Tomasz et al. [21] performed a thorough review of existing research. The literature reveals that a range of machine learning methods (MLMs) and feature selection strategies (FSS) have been employed to identify code smells in these datasets [20–25]. Positive results also show that the majority of these approaches are effective [20, 22, 23]. But most publications don't cover how choosing a certain set of measurements might enhance performance in identifying code smells. The issue of identifying "code smells" has been tackled by deep learning researchers such as Tushar Sharma et al. [24] and Alazba, A. et al. [22]. Evidence has revealed that deep learning and stacked ensemble methods outperform MLTs in terms of performance accuracy. For this reason, we compare various deep learning and ensemble learning methods for identifying code smell. To tackle these issues, developers are increasingly employing machine learning (ML) methods to detect code smells [10]. The most prevalent approach is supervised learning, where a classifier is trained with a group of independent variables (or features) to estimate a dependent variable (such as the existence or severity of a code smell) [11]. Conventional techniques for predicting software quality frequently rely on human code inspection and testing, which can be laborious, arbitrary, and prone to human mistake. It appears that there is a growing interest in using cutting edge technology, such data mining and deep learning, to automate the software quality assurance process. This study offers a novel method for predicting software quality assessed through source-code analysis using data mining and deep learning models. The primary objective of the project is to develop an automated system that leverages source code analysis to accurately predict software quality. Meaningful patterns and features will be extracted from the source code using data mining techniques, capturing both structural and semantic information. Convolutional neural networks (CNNs), a specifictype of deep learning model, would significantly benefit from these extracted features, as they can effectively handle and integrate the complex relationships between the features and software quality metrics.

A large collection of source-code samples and related quality measures will be gathered with the aim of accomplish this goal. The source-code samples will be examined for a diverse set of code metrics, including code complexity, code duplication, and code smells. These measurements, which serve as input features for data mining and deep learning models, will provide insightful opinions about the qualities and attributes of the code. The gathered dataset will go through pre- processing to address any noise or inconsistencies in the data before the model is developed. The most pertinent and instructive elements for software quality prediction will be found using feature selection and dimensionality compression approaches.

In order to guarantee reliable model development and assessment, the pre-processed dataset will thereafter be split into training, validation, and testing sets. Using the training set and the quality measurements and extracted features, deep learning models will be created and trained. For the models to operate at their best and be capable of generalization, optimization procedures such as hyperparameter tuning and regularization will be applied. The validation set will be used to thoroughly assess the trained models, and if necessary, fine-tuning will be carried out to improve their prediction power.

An automated system that uses source code analysis to accurately forecast software quality is the anticipated result of this research. The objective of this strategy is to restore the objectivity and competency of software quality assurance through the use of data mining and deep learning techniques. By enabling early detection of quality concerns, simplifying resource allocation, and eventually enhancing the performance and reliability of software systems, the research findings have the potential to significantly improve the software progress process.

2. Proposed methodology

As illustrated in Figure 1, we develop a model to detect source code problems using ensemble methods. Selecting the appropriate datasets to employ for codsmell detection is the first step. The selected datasets' characteristics are scaled using the min-max normalization technique. For various machine learning techniques, this approach ensures that the features always lie within the same range, which is often 0–1. The Synthetic Minority Over-sampling Technique (SMOTE) is employed to ensure that the datasets accurately represent the classes they are used for. This technique can assist in creating false representations of the underrepresented group in order to address concerns about class imbalance when the code smells are spread unevenly. The Synthetic Minority Over-sampling Technique (SMOTE-NC) includes a variation that integrates both continuous and categorical processing.

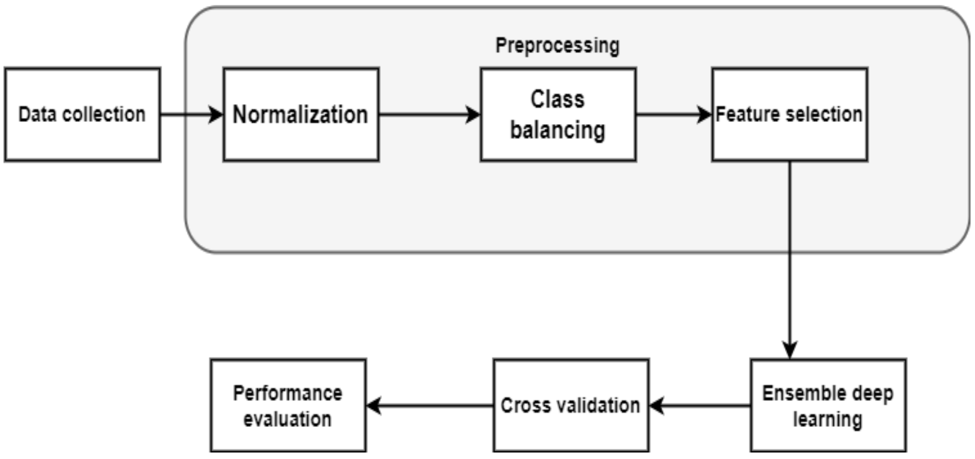


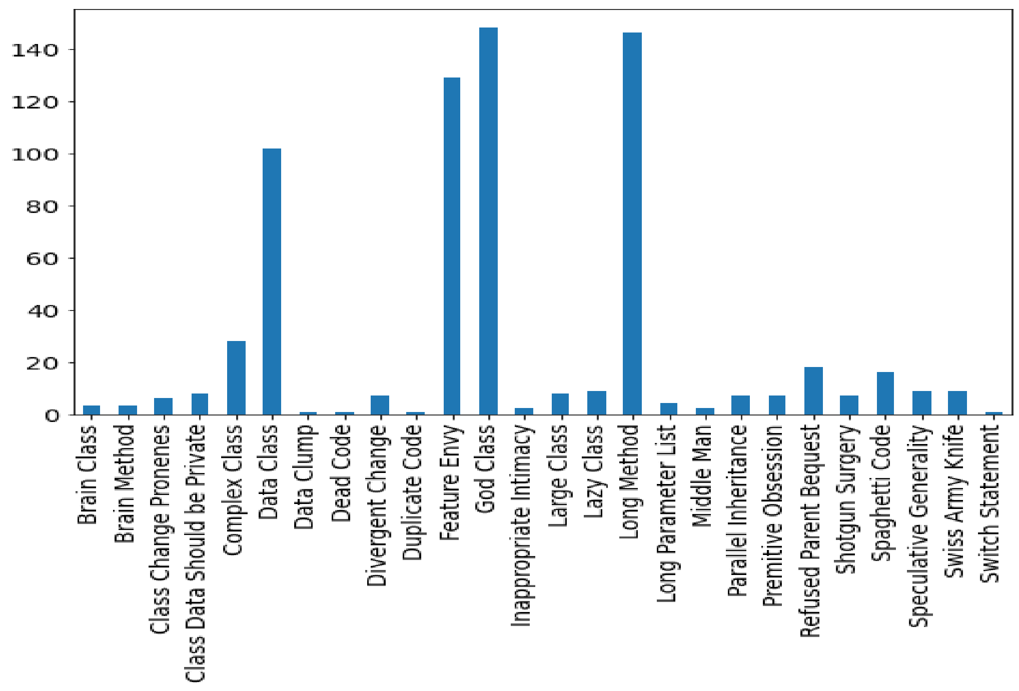
Figure 1. Overall structure of the code-smell detection methods

Employing a recursive approach, RFE reduces features according to weights or relevance until the desired amount of features is reached. Deep learning and ensemble learning models are fed the generated dataset. Ensemble methods aggregate multiple models, whereas deep-learning methods utilize numerous layers of neural-networks for making predictions. Both ensemble and deep- learning methods are enhanced by employing ten-fold cross-validation. This technique partitions the dataset into 10 equal segments (folds), using nine of these segments for training and the remaining one for testing, allowing for repeated training and evaluation. These metrics are subsequently utilized to gauge how effective the ensemble and deep learning methods are. Metrics for code smell detection may include F1-score, recall, accuracy, and precision, among others.

a. Data collection

The dataset in Figure 2 displays the number of different code smells found in a software project available in the Kaggle repository. Each code smell is listed with its respective count of occurrences.

For example, there are 3 occurrences of the Brain Class code smell, which usually points to a class with too much responsibility or complexity and mayneed to be divided into smaller, more manageable, more specialized classes.



Additionally, there are 28 instances of the Complex Class code smell, indicating that a class is overly complex and could benefit from refactoring to make it simpler and more maintainable.

There are 102 instances of the Data Class code smell, which pertains to classes primarily designed to hold data with minimal behavior. This indicates that the class could benefit from either adding more functionality or being refactored to remove excessive data-handling tasks. The dataset also reveals 129 occurrences of the Feature Envy code smell, which occurs when a method in one class excessively relies on data from another class. This often suggests a design flaw, indicating a need to reevaluate the distribution of responsibilities among classes. Additionally, there are 148 cases of the God Class code smell, where a single class becomes overly large and complex, managing too many responsibilities. This usually calls for refactoring to distribute these responsibilities across smaller, more focused classes. Lastly, the dataset reveals 146 cases of the Long Method code smell, where methods become excessively lengthy, making them difficult to understand, test, and maintain. It is generally advisable to split long methods into smaller, more manageable ones.

b. Preprocessing

An extension of SMOTE that works with datasets that have both continuous and categorical features is called SMOTE-NC. By averaging feature values across similar instances, it generates synthetic samples for the underrepresented class.

Algorithm 1. Synthetic Minority Over-sampling Technique for Nominal and Continuous (SMOTE-NC)

```
i. Given:
    1. Minority class instances: $Xminority$
    2. Majority class instances: $X_{\text{majority}}$
    3. Let $Xminority$
    4. denote the feature vectors of the minority class instances.
    5. Let $k$ denote the amount of closest neighbors to consider.
    6. Let $(X, Y)$ denote a a distance measure among two feature vectors $X$ and $Y$.
    7. Let$Xnew$ denote the synthetic sample generated by SMOTE-NC.
ii. Steps of SMOTE-NC:
    1. Select a minority class instance $X$ from $Xminority$.
    2. Find $k$ nearest neighbors of $X$ in $Xminority$based on the distance metric d.
    3. Randomly select one of the nearest neighbor, denoted as $X_{\text{neighbor}}$.
    4. For each feature $f$, calculate the difference between $X$ and $X_{\text{neighbor}}$:
        i. $diff_f = X \cdot f - X_{\text{neighbor}} \cdot f$
    5. End for
    6. Randomly select a value between 0 and 1, denoted as $\lambda$.
    7. Generate the synthetic sample $Xnew$as follows:
        h. $Xnew \cdot f = X \cdot f + \lambda * difff$, where $f$ represents each feature of
        $X$.
i. Repeat steps 1-6 to generate the desired number
```

The synthetic instances are guaranteed to have feature values that align with both continuous and categorical features thanks to SMOTE-NC. By averaging feature values across comparable instances, it creates synthetic samples for the minority class. By doing this, it aims to balance the class distribution, thereby enhancing the dataset's effectiveness for training machine-learning techniques with strong cross-class generalization.

For the purpose of balance the class distribution and address imbalanced datasets, SMOTE-NC creates synthetic samples for the minority class. Recursively removing features based on weights or importance is the objective of RFE feature selection algorithm. Starting with every possible attribute, it progressively removes the less important ones until the desired number of capabilities is reached. It aids in choosing the features that are most relevant to the machine learning activity. RFE helps to improve model performance on imbalanced datasets by ensuring that the features chosen are the most informative and discriminative for class distinction in the situation of unbalanced data. By concentrating on the most discriminative features, RFE is used to identify the most significant features for the machine learning task, which can aid in enhancing model performance on unbalanced datasets. The following is a description of RFE's mathematical notation:

The Firefly Algorithm operates by simulating the natural behavior of fireflies to optimize solutions within a defined search space. Initially, the algorithm sets up parameters such as population size (NP), dimensionality (D), and control parameters like alpha, betamin, and gamma. Firefly agents are randomly positioned within the search space bounded by LB (lower bound) and UB (upper bound). Each firefly's fitness is evaluated using a specified fitness function, determining its quality or suitability as a solution.

Throughout the optimization process, fireflies adjust their positions based on a few key steps. Firstly, fireflies sort themselves based on their light intensity (I), which reflects their fitness. This sorting enables the algorithm to identify

the most promising solutions within the population. Subsequently, fireflies with higher intensities (better fitness) influence the movement of others more strongly. This movement is governed by parameters like alpha, controlling the randomization of firefly movement, beta, which determines the attractiveness between fireflies based on their distances and intensities, and gamma, influencing how quickly attractiveness decreases with distance.
As the algorithm iterates, fireflies replace less fit solutions with better ones, enhancing the population's overall fitness over successive generations. Boundary handling ensures fireflies stay within the predefined search spacelimits (LB and UB). This iterative process goes on until the algorithm completes the specified number of function evaluations (nFES). By harmonizing exploration (randomization) and exploitation (focus on superior solutions), the Firefly Algorithm effectively navigates intricate solution spaces to discover optimal or near-optimal solutions across various optimization challenges. **Alpha Parameter (self.alpha)**: Controls the randomization and movement of fireflies. It oadjusts the randomization factor used in updating firefly positions towards brighter
Algorithm 2. Firefly Algorithm

```
1.  Class algorithm
2.  Initilaze (X, y, n_features):
3.  //Initialization:
4.  Self.D ← X.shape[1], self.NP← 100, self.nFES← 1,self.alpha← 1.0,
Self.betamin← 0.5, self.gamma ← 1
5.  Self.index, self.Firefiles,self.Firefiles_tmp ← [], random intilaization
6.  Self.Fitness, Self.I, Self.nbest←[],[],[]
7.  Self.LB, self.UB, self.fbest, self.evaluations← 0,1,None, 0
8.  Self.X, Self.y, Self.n_features ← X,y,n_features
9.  def alpha_new(a):
10. Return (1-(1.0 –math.pow((math.pow((math.pow(10.0,- 4.0)/0.9),1.0/float(a))) *self.alpha
11. def sort_ffa():
12. self.Index.sort(key=lambda i: self.I[i])
13. def replace_ffa():
14. self.firefiles← [self.Firefiles_tmp[self.Index[i]] for i in range(self.NP)]
15. def FindLimits(k):
16. self.Firefiles[k][i] ← max(self.LB, min(self.UB,self.Firefiles[k][i]))
17. def move_ffa():
18. scale ← abs(self.UB-self.LB)
19. For i in range(self.NP)
20. For j in range(self.NP)
r ← math.sqrt(sum((self.Firefiles[i][k]-self.Firefiles[j][k]) **2
22. For k in range(self.D)
If(self.I[i]<self.I[j])
24. Beta ← (1.0 –self. Betamin)*math.exp(-self.gamma*r*r)+ self.betamin
25. Else
```

```
26. Self.Firefiles[i][k] ← (1.0 –beta) *selfFirefiles [i][k]+beta * self.Firefiles_tmp[j][k]+self.alpha*
    (random.uniform(0,1)-0.5)
27. For k in range(self.D)
28. Return  best firefly position found
```

neighbours during the movement phase (move_ffa function). It calculates a new value using the current alpha value and a function of the iteration count (a), which is handled by the alpha_new function. **Beta Parameter (self.betamin, self.gamma)**: Determines the attractiveness between two fireflies based on their relative brightness. It Influences the amount of movement of each firefly towards other brighter fireflies. It Calculated based on a scaling factor (beta0 - betamin) and an exponential decay function involving the distance between fireflies (r) and a gamma coefficient (gamma). This is implemented in the move_ffa function. **Gamma Parameter (self.gamma)**: Regulates how the attractiveness (beta) between fireflies diminishes based on the distance (r) between them. A higher gamma leads to a quicker decrease in attractiveness with distance, affecting how fireflies are drawn to brighter neighbors. It Higher values of gamma make the algorithm explore more extensively, while lower values may lead to more exploitation of local optima. In the Firefly Algorithm, these parameters collectively determine how fireflies move towards brighter solutions, balancing exploration and exploitation.

The firefly algorithm (FA) is a metaheuristic optimization approach modeled after the light patterns of fireflies. It is particularly effective for addressing optimization problems that traditional methods may struggle with, especially when dealing with intricate search spaces or non-linear relationships. Here's a detailed explanation of the firefly algorithm and its implementation results: In the firefly algorithm, fireflies represent possible solutions to an optimization problem. Each firefly's location in the search space denotes a candidate solution. The goal of the algorithm is to optimize an cost function f(x)f(x)f(x), where xxx denotes its position (or solution) of a firefly within the search space. Fireflies are attracted to others based on their brightness (fitness value). Brighter fireflies are more attractive. The attractiveness III between two fireflies decreases with distance between them. At each iteration, fireflies move towards brighter ones. The movement is governed by parameters: **Alpha (α)**: Randomization parameter affecting the randomness of movement. **Beta (β)**: Light absorption coefficient affecting attractiveness reduction over distance. **Gamma (γ)**: Light intensity coefficient influencing the attractiveness of fireflies. **Optimization Process:** Fireflies adjust their positions iteratively based on their attractiveness and the distance to other fireflies. After evaluating their new positions, fireflies update their brightness (fitness) and attractiveness. Algorithm Iteration: The algorithm persists until a termination criterion is met, such as reaching a specified maximum number of iterations or achieving convergence in the solutions. Recursive Feature Elimination (RFE) systematically removes the least important features according to the evaluations of the trained model, with the goal of pinpointing the most crucial features for the task at hand.

Algorithm 3. Recursive Feature Elimination (RFE)

```
1. Given:
    a. Let $X$ denote the dataset with m instances and $n$ features.
    b. Let $y$ denote the corresponding target variable.
    c. Let $F$ denote the set of all features in $X$.
    d. Let $S$ denote the selected subset of features at each iteration.
    e. Let estimator be the chosen for featureselection.
    f. Let importance $(f)$ represent the importance score of feature $f$ based on theestimator.
2. Steps of RFE:
    a. Initialize $S$ with all features: $S = F$.
    b. Train the estimator on $[S]$ and y and obtain importance scores for each feature.
    c. Rank the features based on their importance scores.
    d. Eliminate the least important feature from $S$.
```

e. If the desired number of features is reached or a predefined stopping criterionis met, stop. Otherwise, go to step 2.
f. End if
g. Return the final subset of selected features S.

c. The selection of an importance measure may vary depending on the specific machine-learning algorithm employed.

Ensemble deep learning

The first CNN classifier is trained on the code smell dataset using convolutional layers to glean useful characteristics from raw data. Similarly, the second CNN classifier is trained on the same code smell dataset using a different architecture or hyper parameter relative to the initial classifier as shown in Figure 3. The objective of using two separate classifiers is to capture different aspects or representations of the code smells, allowing for a more robust anddiverse ensemble. Once both CNN classifiers are trained, their predictions are combined or aggregated to make the final decision.

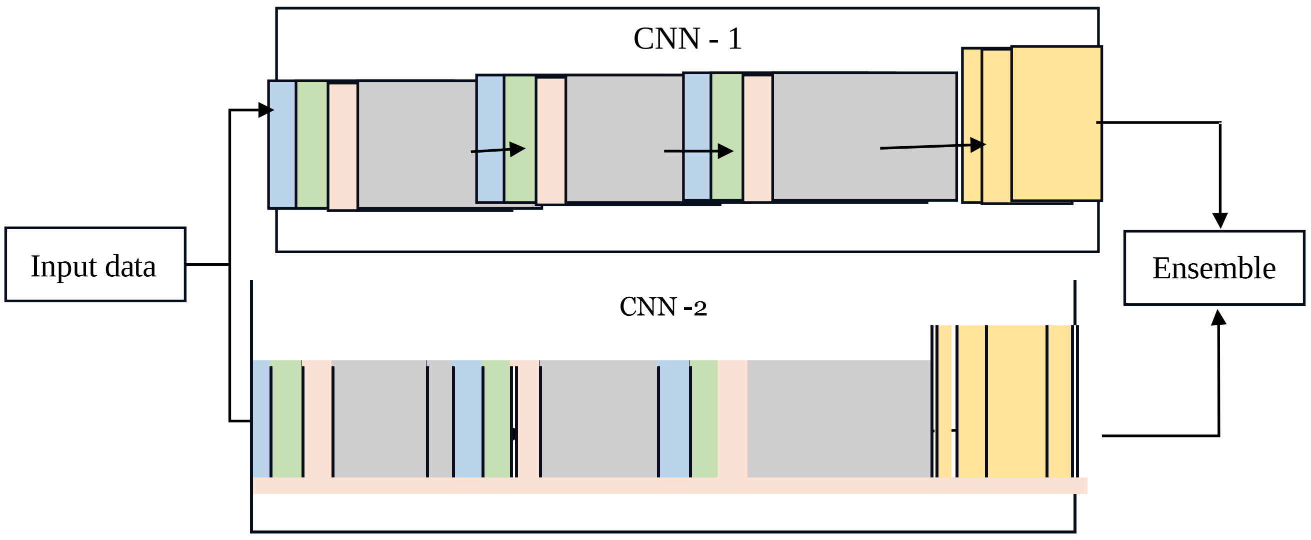


Figure 3. Ensemble Deep learning algorithm

The convolutional layer employs filters to derive elements within the input data. This operation can be represented by the following equation:

$$O_i = \sigma\left(\sum_{j=1}^{N} w_j * I_{i-j} + B_j\right) \quad (1)$$

O_i represents the result of the feature map at position i. $w_j$denotes the learnable weight parameters (filters) of size N used for convolution.I_{i-j} represents the input feature map at position i-j.B_j denotes the learnable bias parameters.σsignifies the activation function (e.g., ReLU, sigmoid) applied element-wise.

The pooling layer subsamples the feature maps, shrinking their spatial dimensions. The equation for a pooling layer can be represented as:

$$O_i = (I_i) \quad (2)$$

O_i indicates the output of the pooling layer at position i.I_i represents the pooling layer at position i.Each neuron in the layer that lies below the completely linked one is linked to every neuron in the layer above it. The formula for a fully connected layer operation written as ,

$$O = (W \cdot I + B) \tag{3}$$

Each classifier predicts the code smell label for a given input, and A majority vote is used to decide the final forecast. For example, if Classifier 1 predicts "Brain Class" and Classifier 2 predicts "Complex Class," the ensemble prediction could be determined as "Complex Class" since it received more votes. On specific requirements, a decision threshold with aggregated predictions to categorize a code smell as present or absent. Forexample, if the aggregated probability for a specific code smell exceeds a certain threshold, it is considered present; otherwise, it is considered absent. The training set consists of raw data representing code segments known to exhibit code smells. Each data instance (X) represents a code segment, and its corresponding label (y) indicates absence. Input (X): Each data instance (X) in data collection is a representation of a code segment. The form of raw code snippets, tokenized sequences, or other appropriate representations suitable for input into the CNN classifiers. Output (y):

Corresponding to each X instance, there exists a label (y) indicating code smell presence. This is a binary classification task where y = 1 denotes the presence of a code smell, and y = 0 denotes its absence.

3. Result and discussion

The dataset comprises 10,000 RGB images, each with dimensions of 256 pixels in width and height. It is organized into five distinct classes, with 2,000 images per class, ensuring a balanced representation across categories. The images depict various real-world objects captured under diverse lighting conditions, contributing to a diverse dataset. The dataset is notably clean, with minimal noise and no missing or corrupted images, making it suitable for training and evaluating machine learning models in image classification tasks. The dataset includes 3D images, which are represented as volumetric data. Each 3D image consists of a grid of voxels (volumetric pixels), typically organized in a 3D array format such as 128x128x128 voxels. Unlike 2D images that capture information in two dimensions (width and height), 3D images add an additional dimension (depth or thickness), providing a spatial representation of objects or scenes. These 3D images are often used in applications where understanding the spatial distribution of features or structures is crucial, such as in medical imaging (like CT scans or MRI), scientific simulations, and computer-aided design (CAD). In dataset, the 3D images likely capture volumetric data of objects or scenes, allowing for analysis and modeling of three-dimensional characteristics.

Incorporating both 2D and 3D images into the dataset enhances its value, allowing scholars and professionals to build and test machine-learning models and techniques across a wider array of applications. This includes everything from conventional image classification to more sophisticated volumetric data analysis and modeling. The efficiency and precision of a code smell detection model may be judged using efficacy measures. The precision of a model may be evaluated by determining what percentage of training data was properly identified. It s defined as,

$$\boldsymbol{Accur} = \frac{Correctly\ classified\ instances}{Total\ number\ of\ instances} \tag{4}$$

Precision evaluates the percentage of correctly identified positive cases out of all cases categorized as positive. It compared to thetotal number of positive results anticipated. It is calculated as,

$$\textbf{Precision} = \frac{True\ \ Positive}{True\ Positive{+}False\ \ Positive} \tag{5}$$

Recall percentage represents the number of true positives that were correctly identified. It is calculated as:

$$\textbf{Recall} = \frac{True\ \ Positive}{True\ Positive{+}False\ \ Negative} \tag{6}$$

The F1 score provides a balanced assessment of both precision and recall. It offers a comprehensive evaluation of a model's performance by considering both these metrics. The calculation is as follows:

$F1 - Score = 2 * 8$ (7)

"Specificity" denotes the ratio of true negative cases accurately identified compared to the overall number of negative cases. It is computed as:

$$\textbf{Specificity} = \frac{True\ negative}{True\ negative + False\ positive} \quad (8)$$

As demonstrated in Table 1, the model reaches complete accuracy, correctly identifying each and every instance of class in code smell. Furthermore, it performs admirably in identifying this code smell, with high precision, recall, and F1-Score values. Furthermore, the model achieves an impressive 99.8% accuracy rate for the Feature Envy code smell. It attains a high accuracy of 99.7%. Excellent results for accuracy and recall also find the model can reliably detect occurrences. The prediction class pertains to the ultimate decision-making phase following the training of several CNN classifiers. In ensemble deep learning, this prediction class involves aggregating or combining the outputs from multiple classifiers to arrive at a unified decision regarding in the input data. Ten-fold cross-validation is a commonly employed method for assessing model performance. Here's an explanation of the rationale behind using ten-fold cross- validation:

1. **Enhanced Evaluation Accuracy:**

Ten-fold cross-validation provides a more dependable evaluation of model performance compared to a single train-test split. This method involves dividing the data into ten folds and using each fold as a test set once while the remaining folds serve as training sets. By averaging the results from

these ten iterations, we reduce the variability in performance metrics like accuracy or error rate.

2. **Utilization of Data:**

By dividing the dataset into ten equal portions (folds).Every data point appears once in the training-set and nine times in the test set. By using this technique, the possibility of bias resulting from a single data division is reduced.

3. **Mitigation of Overfitting:**

By repeating the training with different data partitions, the potential for overfitting to a specific subset is minimized. This approach helps the model generalize more effectively to new, unseen data, which is essential for ensemble and deep learning methods that often handle complex, high-dimensional datasets.

Figure 4.a, showing an accuracy of 100% and a loss of 0.12, depicts the model's accuracy while being trained or evaluation. The accuracy value of 100% shows the proposed system has achieved perfect accuracy on the given dataset or task, correctly classifying all instances. It indicates that the model's predictions align perfectly with the ground truth labels.

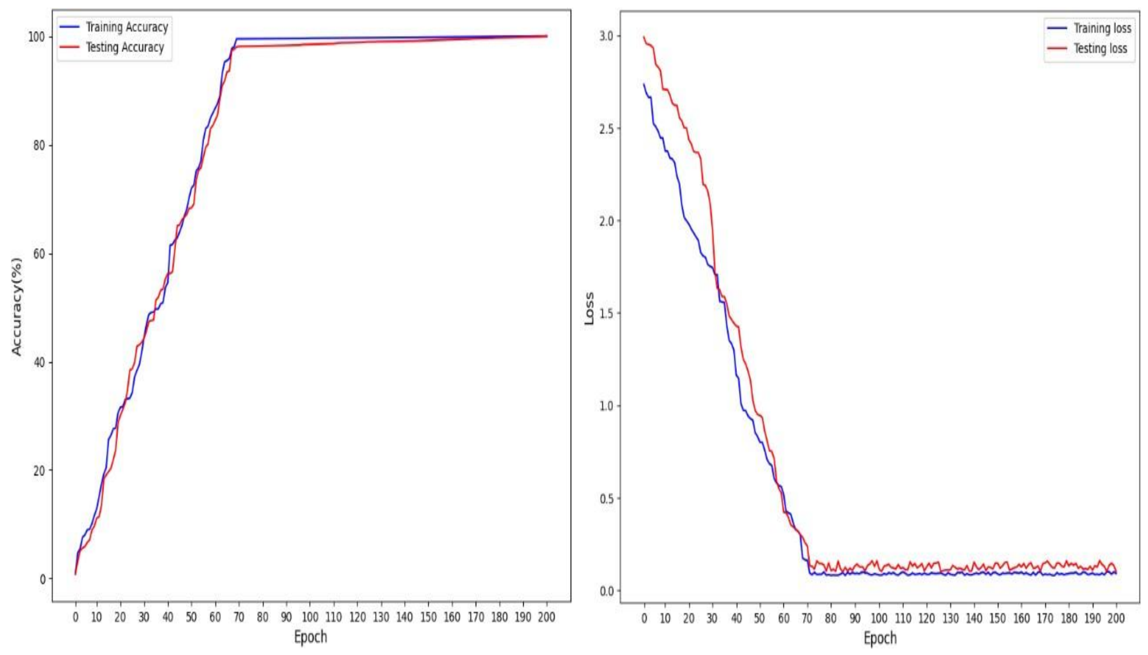
Table 1. The performance analysis of the code class

Class	**Accuracy**	**Precision**	**Recall**	**F1-Score**
Data Class	100%	1.00	0.99	0.99
Feature Envy	99.8%	0.99	0.99	0.99
God Class	99.7%	0.99	0.99	1.00
Long Method	98.9%	0.99	0.98	0.98

Refused Parent Bequest	100%	1.00	1.00	1.00
Spaghetti Code	100%	1.00	0.99	1.00
Others	100%	1.00	1.00	0.99

The loss value of 0.12 from figure 4.b indicates the model's level

of error or how well it fits the training data. A lower loss value indicates a better fit, meaning that Forecasts made by themodel are more in line with the true labels. In this situation, the loss of 0.12 indicates a low degree of inaccuracy in the simulation and is performing well in terms of minimizing the discrepancy between predicted and actual values. While the proposed model for software quality assurance prediction using data mining and deep-learning models presents several advantages. It is crucial to take into account certain limitations of the research. The caliber have a major impact on how well the suggested approach works. The proposed approach heavily relies on extract meaningful features and patterns. However, some software quality issues may not be solely dependent on original source-code itself but can also be influenced by factors such as software design, architectural decisions, or external dependencies. The proposed approach might not fully address these factors, potentially resulting in limitations in accurately predicting software quality.



(a) Accuracy (b) Loss

Figure 4. Accuracy and Loss of the ensemble model on code smell detection

4. Conclusion

In summary, this research introduces a novel approach to software quality

assurance prediction by integrating data mining and deep-learning models. The research methodology involves gathering a comprehensive code-base samples along with their quality metrics. Various code metrics, including complexity, duplication, and code smells, are extracted from dataset and used as input in data mining and deep learning models. Pre-processing techniques are utilized to remove data noise and inconsistencies. Feature-selection and dimensionality reduction techniques are then employed to identify the most pertinent features for software quality prediction. Convolutional Neural Networks (CNNs) are then designed and trained on these features, allowing the models to learn complex relationships between the metrics and software quality indicators. The trained models undergo optimization, including hyper parameter tuning and regularization, to enhance their performance and generalization. The models are evaluated on a validation set and refined as needed to improve their accuracy. Comparing this method to more conventional approaches like manual code examination and testing reveals many benefits. By extracting structural and semantic information from the source code, it makes software quality assurance more automated and effective while producing predictions that are more trustworthy and accurate.

References

[1] M. Fowler and K. Beck, Refactoring: improving the design of existing code, Second Edi. Addison-Wesley, 2018. [Online].
Available: https://refactoring.com/

[2] M. I. Azeem, F. Palomba, L. Shi, and Q. Wang, "Machine learning techniques for code smell detection: a systematic literature review and meta-analysis," Inf Softw Technol, vol. 108, pp. 115–138, Apr. 2019, doi: 10.1016/j.infsof.2018.12.009.

[3] D. di Nucci, F. Palomba, D. A. Tamburri, A. Serebrenik, and A. de Lucia, "Detecting code smells using machine learning techniques: are we there yet?," in 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), Mar. 2018, pp. 612–621. doi: 10.1109/SANER.2018.8330266.

[4] F. Pecorelli, D. Di Nucci, C. De Roover, and A. De Lucia, "A large empirical assessment of the role of data balancing in machine learning-based code smell detection," Journal of Systems and Software, vol. 169, p. 110693, 2020, doi: https://doi.org/10.1016/j.jss.2020.110693

[5] Fontana, F.A.; Zanoni, M. Code smell severity classification using. machine learning techniques. Knowl. Based Syst. 2017, 128, 43–58

[6] Pushpalatha, M.N.; Mrunalini, M. Predicting the Severity of Closed Source Bug Reports Using Ensemble Methods. In Smart Intelligent Computing and Applications. Smart Innovation, Systems and Technologies; Satapathy, S., Bhateja, V., Das, S., Eds.; Springer: Singapore, 2019; Volume 105

[7] Sharma, Rohini. "Flow based anomaly detection and behavioral analysis in computer networks.", Springer, 2020.

[8] Baarah, A.; Aloqaily, A.; Salah, Z.; Mannam, Z.; Sallam, M. Machine Learning Approaches for Predicting the Severity Level of Software Bug Reports in Closed Source Projects. Int. J. Adv. Comput. Sci. Appl. 2019, 10, 285–294.

[9] Pushpalatha, M.N.; Mrunalini, M. Predicting the severity of open source bug reports using unsupervised and supervised techniques. Int. J. Open Source Softw. Process. 2019, 10, 676–692.

[10] Dewangan, S.; Rao, R.S.; Mishra, A.; Gupta, M. A Novel Approach for Code Smell Detection: An Empirical Study. IEEE Access 2021, 9, 162869–162883

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

[12] E. Alpaydin, Introduction to machine learning. MIT press, 2014.