Received: 9 January 2022 | Revised: 17 October 2022 | Accepted: 8 November 2022

DOI: 10.1002/cpe.7531

RESEARCH ARTICLE

WILEY

Software code refactoring based on deep neural network-based fitness function

Chitti Babu Karakati[1] | **Sethukarasi Thirumaaran**[2]

[1]Department of Information Technology, R.M.K. Engineering College, Kavaraipettai, Tamil Nadu, India

[2]Department of Computer Science and Engineering, R.M.K. Engineering College, Kavaraipettai, Tamil Nadu, India

Correspondence
Chitti Babu Karakati, Department of Information Technology, R.M.K. Engineering College, Kavaraipettai, Tamil Nadu, India.
Email: mailtocbabu@gmail.com

Summary

Refactoring is extensively recognized for enhancing the internal structure of object-oriented software while preserving its external behavior. However, determining refactoring opportunities is challenging for designers and researchers alike. In recent years, machine learning algorithms have shown a great possibility of resolving this issue. This study proposes a deep neural network-based fitness function (DNNFF) to resolve the software refactoring issue. This study suggests an effective learning technique that automatically featured extracted from the trained models and predicted code clones to recommend which category to refactor. The software engineers automatically assess the recommended refactoring solutions using Genetic Algorithms (GA) for minimum iterations. A Deep Neural Networks (DNN) utilizes these training instances to assess the refactoring solutions for the residual iterations. The refactoring process primarily depends on software designers' skills and perceptions. The simulation findings demonstrate that the suggested DNNFF model enhances the code change score of 98.7%, automatic refactoring score of 97.3%, defect correlation ratio of 96.9%, refactoring precision ratio of 95.9%, flaw detection ratio of 94.4%, and reduces the execution time of 10.2% compared to other existing methods.

KEYWORDS

deep neural network, genetic algorithm, machine learning algorithms, software code refactoring

1 | INTRODUCTION TO SOFTWARE CODE REFACTORING

Refactoring is a set of program transformations envisioned to enhance the system design while conserving the desired behavior, becoming a vital software maintenance action, particularly with the rising complexity of software systems.[1] Software Refactoring involves changing the application's internal structure or design without changing its external behavior or functionality.[2] The semantics of the code before and after refactoring remain the same. Refactoring benefits software development by making programs more understandable and easily discovering software bugs quicker.[3] For software to be useful, it must constantly change. The technology's initial design changes as it adjusts to the new surroundings.[4] The complexity of the code increases due to modifications and enhancements, which lowers the system's performance. These variables increase the time and expense of software maintenance.[5] Keeping the source code is one of the most demanding when developing and evaluating software. Changes are unavoidable for software to stay current, useful, and high-quality.[6] When the structure of a code element is mistakenly anticipated, or the response by the programmers is wrongly conceived, bad odors are generated in a software system.[7] A code smell indicates a more serious problem in technology. Because they show a flaw in the software's architecture that could result in a future breakdown, they are not defects; nonetheless, they can be legally accurate.[8]

Manually inspect source codes to determine refactoring opportunities.[9] Multiple tools for an automatic suggestion of refactoring prospects have been presented.[10] In fully automated refactoring, designers offer their code as input, and the tool automatically gives refactoring recommendations.[11] The mainstream of existing automated refactoring tools assumes that designers want to fix code smells.[12] The refactoring method can be used to identify and correct code odors. Refactoring can be used to improve poorly designed code and turn it into something usable.[11] Refactoring can enhance the technology's exterior characteristics, such as renewability, maintainable code, and legibility.[13] Understanding the most common code smells, the most common refactoring strategies, and the program measurements that strongly affect them is essential to improving software quality.[14] In this situation, reformation is required to change legacy or deteriorated code into a more structured or modular form or migrate code to the various programming languages or language models.[15]

The users usually define the rules manually, a tedious and challenging task.[16] Learning from data utilizing machine learning algorithms can support extract regulations, and for this method to succeed, a substantial amount of information must be fed into algorithms it can learn.[17] Likewise, the search-based algorithm aims to identify a bad smell based on the values of pre-described metrics.[18] The introduction of machine learning algorithms that spontaneously features extracted from the predicted code clone.[19] Then trained models to recommend designers on the refactored clone codes categories and those that are not refactored.[20] A system based on deep learning that recommends the essential clone for refactorings could be useful.[21] Refactoring opportunity prediction has been defined as a binary classification issue and deployed deep learning algorithms to resolve it.[22] The models categorize a particular refactoring type and a regular class utilizing the metrics as features. Numerous deep learning experiments have been intended based on an empirical study of the refactorings.[23]

The main influence of the article is as follows:

- Designing the DNNFF model for software code refactoring using a genetic algorithm.
- Evaluating the mathematical model of genetic algorithm and deep neural network for software code refactoring.
- The investigational outcomes have been completed. The suggested DNNFF model enhances the code change score, automatic refactoring score, defect correlation ratio, refactoring precision ratio, flaw detection ratio, and execution time related to other existing models.

The rest of the article is arranged: Section 2 discusses the related works on software code refactoring. In Section 3, the DNNFF model has been suggested, and in Section 4, simulation analysis has been performed. Finally, Section 5 concludes the study article.

2 | RELATED WORKS ON SOFTWARE CODE REFACTORING

Nasagh et al.[24] proposed the fuzzy genetic automatic refactoring approach (FGARA) to enhance software maintainability and flexibility. A graph model is utilized as the core depiction scheme and measures load, betweenness, out-degree, in-degree, and closeness centrality to determine the program's code smell. Then, the employed fuzzy method is joined with the GA to refactor the code utilizing the graph-related features. It is exposed that the suggested method can recognize 68.92% of the bad classes like the Fontana dataset and refactors 77% of the types properly concerning the coupling processes. This is a remarkable result between the existing refactoring process and studies considering bad smells identification and refactoring. Sharif[25] found extract class refactoring in Vb.Net classes using the Code Analysis Tool (CAT). Thus, this article aims to create a tool that aids developers in spotting potential refactoring opportunities. Both refactoring actions and the refactoring proposed methodology are taken into account. They concentrated on Extract Class when it came to refactoring. Predicting the likelihood of code reworking is possible using object-oriented metrics. Two refactoring factors are combined to indicate the possibility of refactoring efforts and to identify the classes concerned with those efforts.

The tool recognized 11 out of 14 results, giving them an accuracy rate of 79%. Besides assisting software developers in enhancing their code, this research could provide additional information about how refactoring enhances systems. Mohan and Greer[26] discussed the Many-Objective Approach (MOA) for Investigating Automated Refactoring. Experiments have been built to compare the many-objective method against a mono-objective method that only utilizes a single objective to measure the quality of the software. Diverse permutations of the goals were compared and tested to see how well the other goals could work together in a multi-objective refactoring method. The eight methods are verified on six various open-source Java programs. The many-objective method gives a better objective score than the mono-objective method and in less period.

However, the priority and component recentness purposes are less effective in many/multi-objective setups when used together. Meng and Su[27] deliberated the worst-case execution time (WCET) optimization strategy based on source code refactoring. The framework leads programmers through an appropriate sequence to look for refactoring possibilities. The worst-case execution path (WCEP) is first removed from a control flow chart of the target program in this respect. The WCEP is mapped using the back annotation approach to the source code. An abstract syntax tree-based invariant path identification technique is created to recognize the continuous pathways from the WCEP source level. According to

guidelines and loop statements, the source code is split into four optimization zones with consistent priority. This reduces the search scope and prevents incorrect refactoring. This is the foundation on which refactoring is initially carried out, which in the same location has the lowest costs. A cost model of source code refactoring is developed to support the technique.

Based on the above survey, there is a lack of proper tools in existing models. Hence, this paper has suggested the DNNFF model using a genetic algorithm. The subsequent section deliberates the proposed DNNFF model briefly.

3 | DEEP NEURAL NETWORK-BASED FITNESS FUNCTION (DNNFF)

Code refactoring is restructuring computer codes without adding or changing functionality and external behavior. There are several ways to refactoring, and it most often encompasses applying a sequence of standardized, basic activities, known as micro-refactorings. Refactoring is a process to enhance the existing code quality. It operates by employing a sequence of small stages, every of which changes the internal structure of the code while conserving its external behavior. Our proposed DNNFF method proceeds as input to the system to refactor a comprehensive list of probable refactoring categories and the number of developers' connections during the search progression. It produces as output the better refactoring series that enhances the system quality. Our method comprises two major elements: the learning module (LGA) and the interactive component (IGA). The algorithm starts initially by performing the interactive (IGA) component, where the developer assesses the refactoring resolutions manually produced by genetic algorithms for several iterations. The developer assesses the viability and the quality/efficacy of the recommended refactoring one by one since every refactoring resolution is a series of refactoring operations. Therefore, based on his inclinations, the developer classifies all the recommended refactorings as better or not. After several iterations of the interactive (IGA) element, every developer's assessed solutions are reflected training set for the second element, LGA of algorithms.

Figure 1 shows the model refactoring process. The learning genetic algorithm module component executes a DNN to produce a predictive model to estimate the refactoring solutions' evaluation in the following iteration of the genetic algorithm. Therefore, our method does not require the description of fitness functions. The DNN was the best accurate predictor in this investigation, even when dealing with noisy and poor data. There are multiple layers to the structure, and the weights of each associate have been set at random at the beginning of training. As far as artificial neurons go, sigmoid functions suffice with continuous data. Networks can be trained by using an iterative learning method. Two factors determine how well it works. First, momentum factors prevent local minima by reducing the weights of the system. In addition, the learning ratio affects how quickly weight loss occurs.

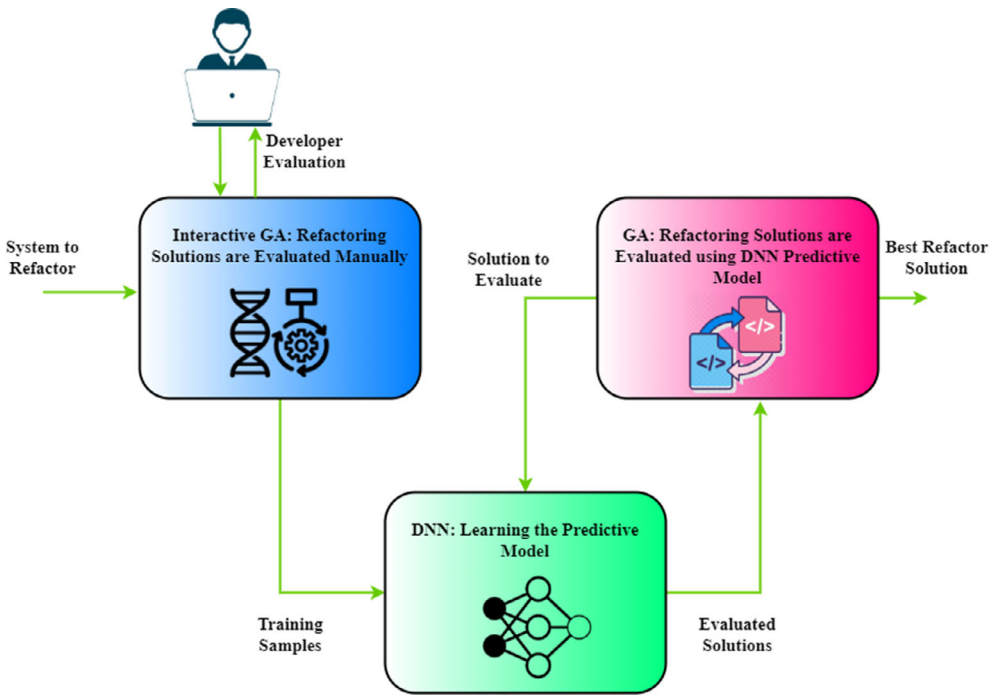

FIGURE 1 Model refactoring process

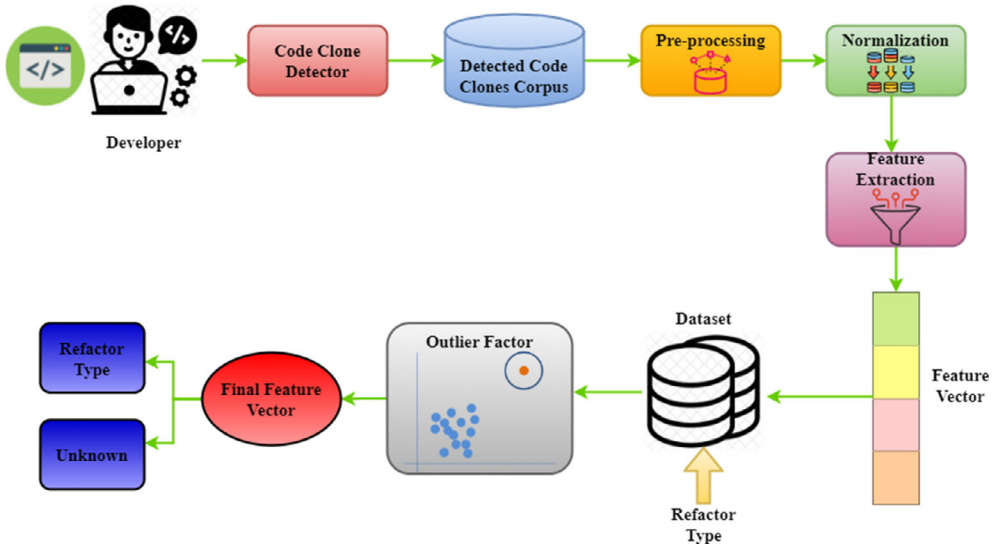

FIGURE 2 Proposed DNNFF model

Figure 2 shows the proposed DNNFF model. This article deliberates the general and new classification algorithm for outlier prediction models and refactoring code clones classification models to classify the test instances as unknown or recognized class sizes. Our classifier models' enhanced efficiency depends upon their locked set validations. Model validation in deep learning is the progression whereby train models are assessed with the testing dataset. The testing datasets in closed set validations comprise instances owned by known classes. This study ran outlier algorithms for the dataset to determine the data point with inconsistencies or differences considerably from the other provided data point. Formerly, the data point class is altered into an unidentified class. After predicting outlier data points, this study builds our closed-set classification model and analyses their efficiency after training: this study trains and tests our classifier with a vector of the dataset. The outlier factors are the abnormality score of every sample. It predicts outliers about their adjacent utilizing k-nearest neighbors (KNN) for approximating the confined density. It predicts subordinate density than neighbors, deliberates outliers, and modifies the unidentified class to datasets. Our model is a stage-by-stage method, and first, this study standardized every source code fragment into special token series. Furthermore, syntax analysis and existing lexical tools transform source code fragments into abstract syntax trees to predict each block from the given source file. In conclusion, extracted features from every code fragment utilizing the Java Development Tools. Then, this study formed pairs of examples by the feature vector from the actual data. This study feeds the feature vector of two target blocks to the feature vector datasets and passes it via confined outlier element algorithms for outlier prediction.

Figure 3 shows the refactoring recommendation model. Deep learning is empathetic to the fundamental learning principles via computer science, statistics, and mathematics. It is a division of artificial intelligence that aims to construct statistical models based on information from famous training groups for making predictions or decisions without being unambiguously programmed. Deep learning comprises several sub-fields and applications: neural networks, statistical learning approaches, data mining, example-based learning, genetic algorithm, natural language processing (NLP), image recognition, and improved learning. This study adopts a deep learning algorithm to construct a detection model for identifying refactored categories. For every classifier, the default configuration and variables are recommended. Our system has two stages, testing and training, like any other deep learning model. This study uses marked pairs of refactored codes from a provided corpus in training. This study trains and tests the model utilizing classification models, from the popular KNN and Bagging model to a newly published class execution, solving classification and regression problems. Forest by Penalizing Attributes is a decision forest algorithm constructing sets of precise decision trees utilizing the powers of every non-class attribute in datasets and concurrently impressive penalty (inconvenient weight).

Data used to train an application or machine learning approach to simulate the desired outcome is training data. The algorithm analysts employ to train the machine can be evaluated based on the accuracy and efficiency measured by the testing data. It is predefined data that is used to compare with the analytical results.

In this context, "Refactoring" refers to rearranging computer code without altering the code's exterior functions and behaviors. Refactoring can take several forms, but the most common one is the application of a succession of standardized, fundamental activities, which are frequently referred to as micro-refactoring. Many social media applications use PHP, and anyone can refactor the software coding using the proposed technique without changing the actual output.

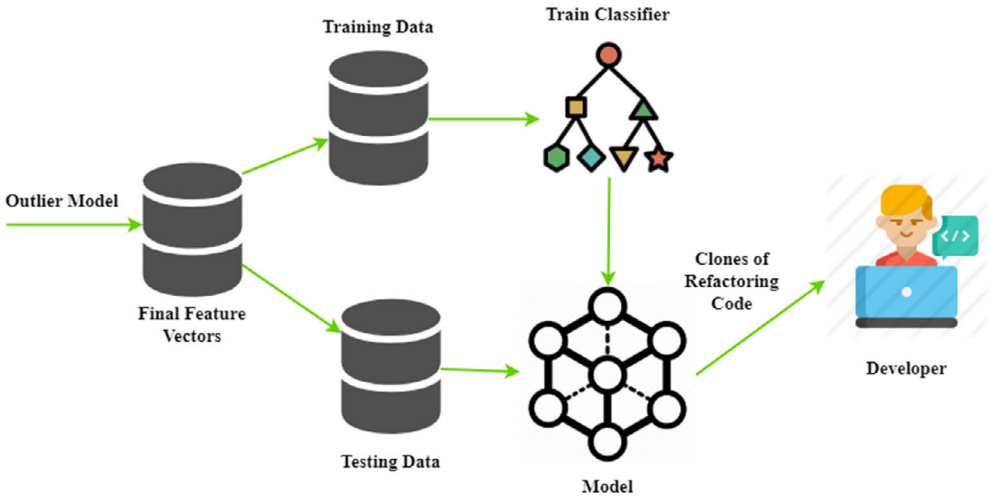

FIGURE 3 Refactoring recommendation model

The data used in the training dataset must be consistent, which used to be part of the learning process. This study uses the Min-max approach because it is one of the most accurate approaches. This study utilized the subsequent data depiction of the genetic algorithm-based learning issue utilizing DNN for software refactorings. Let us signify by D the training set of the DNN. It is self-possessed of a set of couples that symbolize the refactoring series and its assessment.

$$D = \{(Y_1x_1), (Y_2x_2), (Y_3x_3), \ldots (Y_lx_l), \ldots (Y_mx_m)\}, \quad l \in [1, m]. \tag{1}$$

As shown in Equation (1) where, Y_l denotes the refactoring sequence signified as $Y_l = [y_{l1}, y_{l2}, \ldots y_{lt}, \ldots y_{lq}]$, $t \in [1, q]$, x_l indicates the assessment linked with the lth refactoring series in the range $x_l \in [0, 1]$.

Let us indicate by P the matrices that involve numerical value connected to the set of refactoring and by X the vectors that comprise numerical value demonstrating x_l assessments. P is composed of m line and q column where m is equivalent to the number of refactoring series and q is equivalent to the number of solutions.

$$P = \begin{bmatrix} y_{11} & y_{12} & \cdots & y_{1q} \\ y_{21} & y_{22} & \cdots & y_{2q} \\ \cdots & \cdots & \ddots & \vdots \\ \cdots & \cdots & \ddots & \vdots \\ \cdots & \cdots & \ddots & \vdots \\ y_{m1} & y_{m2} & \cdots & y_{mq} \end{bmatrix} X = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ \vdots \\ \vdots \\ x_m \end{bmatrix}. \tag{2}$$

The probability $O(r_j/C_i)$ for particular refactoring types r_j to be chosen when a particular code smell C_i is linked with the chosen element.

- Let $N_m(C_i, r_j)$ be the overall number of refactoring types r_j that is linked with code smell C_i and
- Let $N(C_i)$ be the overall number of refactorings linked with C_i, then

The probability of $O(r_j/C_i)$ is computed as follows:

$$O(r_j/C_i) = N_m(C_i, r_j)/N(C_i). \tag{3}$$

An initial survey was conducted to determine which source code element will be used for each refactoring type. Second, all refactorings involving the identical source code element are clustered together. It was determined how likely every given refactoring with a given sort of source code will be used since each refactoring has its component. Refactorings are always planned when a certain piece of source code has been chosen as the focus of a particular effort. These calculations have been done as follows:

- Let $M_{j,i}\left(e_i, r_j\right)$ be the number of respondents who chosen the specific source code component of the type e_i for particular refactoring types r_j.
- Let $M_j\left(e_i\right)$ be the number of respondents who chosen the specific source code component of the type e_i to employ refactorings.
- Let $O\left(r_j, e_i\right)$ be the probability of employing specific refactoring types r_j while the source code component of types e_i is chosen.

$$O\left(r_j, e_i\right) = M_{j,i}\left(e_i, r_j\right) / M_j\left(e_i\right). \tag{4}$$

Technique name is the most frequently refactored source code element and thus makes it the most common source code pick while refactoring, according to this study. Based on the feedback to a survey, this report forecasts which refactorings should be placed first on a refactoring menu. Source code components and code smells are used to make the forecasts. This study examines how a code smell affects a person's decision-making process. Code smell has been linked to the refactorings in the survey since the first refactoring was discovered. The computations have been carried out as follows:

- Let $M\left(e_i, C_i, r_j\right)$ be the overall number of chosen source code elements of types e_i when the source code is associated with code smells of types C_i and the employed refactoring type r_j.
- Let $M\left(e_i, C_i\right)$ be the overall number of chosen source code elements of types e_i that is linked with code smells of types C_i.

The probability of a refactoring r_j to be ranked on top of the refactoring menu will be provided by:

$$M\left(e_i, C_i, r_j\right) = M\left(e_i, C_i, r_j\right) / M\left(e_i, C_i\right). \tag{5}$$

The quality criterion is assessed utilizing fitness functions provided in expression 6. When the number of code defects decreases after refactorings, the value of quality increases. This method returns to the number of defects predicted earlier refactorings and the number of design flaws after refactoring (detected utilizing the rules on smell detection).

Figure 4 shows the flow chart of refactoring stages on code quality. This research aims to measure the influence of refactoring on code quality by utilizing different internal and external software quality attributes to improve its quality. The study uses external and internal attributes to attain the above objective. Most of the earlier research concentrated on either external or internal attributes. Therefore, this research primarily concentrates

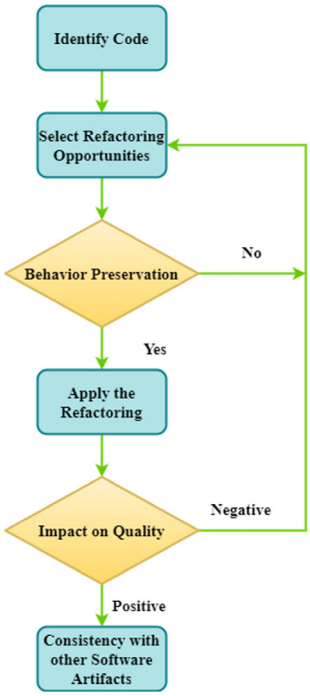

FIGURE 4 Flow chart of refactoring stages on code quality

```java
public class NumericValues {
        private int a;
        private int b;
        public int getA() {
            return a;
        }
        public void setA(int a) {
            this.a = a;
        }
        public int getB() {
            return b;
        }
        public void setB(int b) {
            this.b = b;
        }
        public NumericValues(int a, int b) {
            super();
            this.a = a;
            this.b = b;
        }
 // Find Maximum value
        public int max(int a,int b){
            if(a>b){
                return a;
            }
            else{
                return b;
            }
        }
 //     Find Minimum value
        public int min(int a,int b){
            if(a<b){
                return a;
```

FIGURE 5 Example: Simple NumericValues class

on measuring both scales distinctly to measure the effect of the refactoring on the code quality. Internal quality attributes are utilized to improve the impact of code refactoring on software quality. To calculate the effect of refactoring on external quality characteristics, the five quality attributes utilized in this research are reusability, maintainability, efficiency, understandability, and performance. The following stages show the procedure of refactoring that is employed to measure the effect of software refactoring on both external and internal quality attributes for improving the code quality: Determining the code that must be refactored, identifying the refactoring methods that be employed, Undertaking that the applied refactoring techniques maintain the behavior of software after refactoring, Applying the selected refactoring methods, Assessing the influence of refactoring methods on both external and internal software quality attributes, Keeping the determination between the software that has been refactored and other software configurations.

A simple NumericValues class is taken as example as in Figure 5 to refactor using the proposed approach. The NumericValue class takes two integer numerals as input and performs different operations like finding maximum, minimum, and so forth. The refactored version of the method max is shown in Figure 6. Predicting defects depends on the metrics-based rule in line with which code fragments can be categorized as design defects or not (without likelihood/risk scores), that is, 0 or 1. The defect correction ratio is described by:

$$\text{DCR} = 1 - \frac{\text{Defect after applying refactorings}}{\text{Defect befor applying refactorings}}. \tag{6}$$

Refactoring Operation (RO) are categorized into two categories: High-Level RO (HLR) and Low-Level RO (LLR). A high-level RO is a series of two or more RO. A low-level RO is a basic refactoring containing one basic ROs. The weight ω_j Every RO is an integer value in the array [1, … , 3] reliant on code fragments' difficulty and change effect. For refactoring solutions containing p ROs, the code changes scores are calculated as:

$$\text{code changes} = -\sum_{j=1}^{p} \omega_j. \tag{7}$$

This study defines the following function to calculate similarity scores among suggested refactoring operations and recorded code change:

$$\text{Similarity refactoring history (RO)} = \sum_{i=1}^{m} e_i. \tag{8}$$

As shown in Equation (8), where, m denotes the number of the documented refactoring operation employed to the model in the past, and e_i indicates refactoring weights that redirects the similarity among the recommended refactoring operations (RO) and notes refactoring operations i.

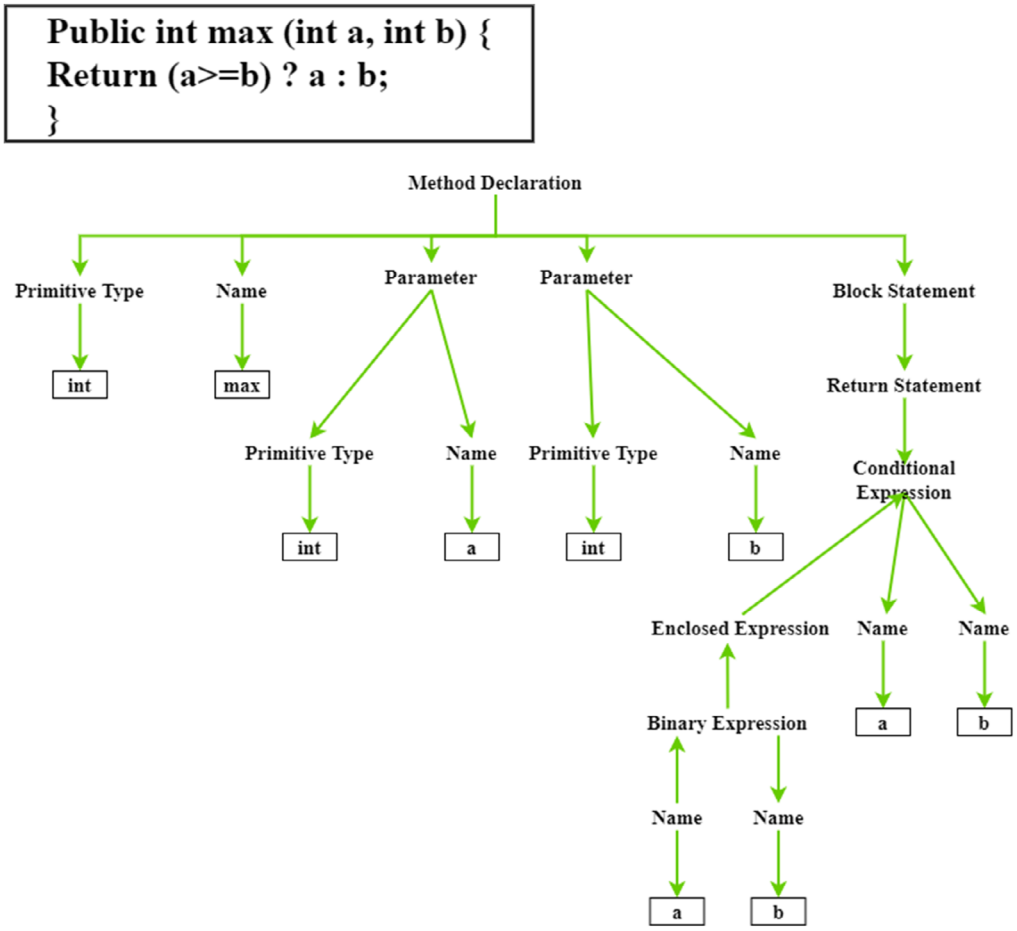

FIGURE 6 Path representation of code

This study calculates the semantics similarity among actors utilizing a data retrieval-based method, explicitly cosine similarity, as illustrated in expression 9. Every actor is signified as *m*-dimensional vectors, where every dimension relates to vocabulary terms. The cosine of the angle among two vectors has been measured as a pointer of similarity. Utilizing cosine similarity, the conceptual likeness among two actors c_1 and c_2 is identified by:

$$\text{Sim}\ (c_1, c_2) = Cos\ (\vec{c}_1, \vec{c}_2) = \frac{\vec{c}_1 \cdot \vec{c}_2}{\|\vec{c}_1\| \times \|\vec{c}_2\|} = \frac{\sum_{j=1}^{m} \omega_{j,1} \times \omega_{j,2}}{\sqrt{\sum_{j=1}^{m} \omega^2_{j,1}} \times \sqrt{\sum_{j=1}^{m} \omega^2_{j,2}}}. \tag{9}$$

As discussed in Equation (9) where, $\vec{c}_1 = (\omega_{1,1}, \ldots \omega_{m,1})$ denotes term vectors respective to actors c_1 and $\vec{c}_2 = (\omega_{1,2}, \ldots \omega_{m,2})$ is term vectors respective to c_2. The weights $\omega_{j,i}$ can be calculated utilizing data retrieval-based methods like the term frequencies.

Figure 6 shows the Path representation of the code. To use machine learning techniques with source code, we must first obtain a numerical representation of the code fragments. To accomplish this, code2vec, a neural technique that learns distributed vectors (embeddings) of code fragments based on their syntactic structure and semantic information captured in method and identifier names, was used. This study first must get a numerical depiction of the code fragment to use deep learning methods when working with source codes. First, the method parses a code snippet into an Abstract Syntax Tree (AST) and extracts a syntactic path between every leaf node negotiating via their lowermost ancestor. Every path is denoted as a series of intermediary Abstract Syntax Tree nodes among two leaf nodes, and the arrow demonstrates the Abstract Syntax Tree route. The whole code snippet has measured a bag of path settings. The system learns numeric vectors for every element of path contexts (i.e., two leaf nodes and a path among them). Then three vectors are concatenated into a single vector named pooled context vectors. To aggregate the data from every path context into one vector representing the whole code fragment, the method employs the attention mechanisms that compute a weighted mean overall united context vector by allocating higher weight to the path that seizure more significant semantic data code fragments. Path-context is defined as a tuple of two nodes and a corresponding path between them. For example, path-context from Figure 5 is represented as (a, Name ↑ BinaryExpression ↑ EnclosedExpression ↑ ConditionalExpression ↓ Name, a).

The entire code snippet is regarded as a collection of path-contexts. The model learns a numeric vector for each component of a path-context (i.e., two leaf nodes and a path between them), and then three vectors are concatenated into a single vector called combined context vector. The approach employs the attention mechanism, which computes a weighted average over all combined context vectors by assigning greater weight to paths that capture more important semantic information for this code fragment, to aggregate the information from each path-context into one vector that represents the entire code fragment. The 384-bit fixed length code vector that results can then be used in the prediction task. The authors limit the length (number of nodes) and width (maximum allowed difference between sibling nodes) of paths to reduce sparsity and the amount of training data. The resultant fixed-length code vectors of size can be additionally utilized in the forecasting task. This study limits the length and paths "width to decrease the amount of training information and sparsity" (i.e., maximally permissible variance among sibling nodes). Based on these embeddings, a system has been constructed to detect techniques terms from their bodies utilizing datasets. By definition, refactoring is the process of rearranging the internal structure of a program without modifying its external behavior or features based on a Deep Neural Network. Refactoring can be approached in various ways, although it typically involves carrying out a set of predefined, elementary procedures, also known as Software Code Refactoring.

Consequently, the code2vec model learned to generate embeddings so that similar code snippets are allocated similar embeddings. The proposed model suggests that these embeddings can be utilized for other tasks where this semantic similarity can be beneficial, like code summarisations, refactoring recommendations, and code searches. The proposed DNNFF increases the code change score, automatic refactoring score, defect correlation ratio, refactoring precision ratio, flaw detection ratio, and execution time related to other models.

4 | SIMULATION ANALYSIS

The study aims to assess our refactoring tools' effectiveness and usefulness in practice. This study showed a non-subjective assessment with potential designers who can use our refactoring tools.

4.1 | Environmental setup

The proposed approach was tested using the hardware configuration of Intel Core i7 processor with 8 Gb RAM, 512 Gb SSD and software setup of Microsoft Windows 10 64-bit operating system, Python 3.7 with TensorFlow 2.1. Empirical evaluation will be done using open-source software models in this work.[28] Let us now describe the datasets used to evaluate our methodology.

4.2 | JMove's dataset

In this section, we evaluate and compare our approach to existing refactoring recommendation tools. We decided to test our approach on two datasets: in the first part, we test it on the manually prepared dataset used in Reference 29, and in the second part, we test it on a synthetic dataset. In both cases, we use projects that were not encountered during the training phase and compare our results to those of the existing recommendation tools, JDeodorant[29] and JMove.[29] They were chosen for comparison because they are publicly available and represent the state-of-the-art in this field. JDeodorant is also commonly used as a reference in prior refactoring recommendation research. The following steps comprise the evaluation. Using MoveMethodGenerator, we search for movable methods and possible target classes in each project in our datasets, as described in below.

We evaluate the proposed method on seven open-source projects from Table 1 of JMove's dataset. The table includes the project's name, version, number of classes (NOC), number of methods (NOM), and number of source code lines (LOC).

4.3 | Synthetic dataset

To minimize the impact of a manual analysis of samples, we also evaluate our method using a synthetic data set. We extracted five high-quality open-source projects from the publicly available synthetic dataset MoveMethodDataset, which was generated by the MoveMethodGenerator tool. The projects utilized for this dataset are presented in Table 2 and differ from those used to train the model.

In the software development industry, these technologies have a wide range of capabilities and a high level of maturity. Code change score, automatic refactoring score, defect correlation ratio, refactoring precision ratio, and flaw detection ratio are outcome measures used to evaluate the suggested DNNFF model.

TABLE 1 Subject applications from JMove's dataset

Application	Version	NOC	NOM	LOC
Weka	3.6.9	908	16,034	257,897
ant	1.8.2	760	8586	103,402
Freecol	0.10.3	535	6616	93,605
Jmeter	2.5.1	682	7392	81,222
Freemind	0.9.0	368	4074	53,782
Jtopen	7.8 1	1450	22,143	340,752
maven	3.0.5	154	1568	71,065

TABLE 2 Subject applications from the synthetic dataset

Application	Version	NOC	NOM	LOC
Pmd	6.13.0	1147	8637	119,430
Cayenne	4.2 1	499	12,164	275,450
Pinpoint	1.9.0	2551	17,024	290,974
Jenkins	1.51	768	6292	155,667
Drools	7.22.0	2758	27,793	680,234

i Code change score

Various code changes are made in the application of refactorings. The number of codes that are changed relates to the number of codes that have been updated by adding, erasing, or moving operations (e.g., classes, procedures, fields, relationships, field references, etc.). Code changes must be minimized when proposing refactoring to save effort and assist developers in comprehending modified/improved design. Most developers aim to use the original design structure to address design faults as much as feasible. Therefore, it is important to reduce code changes and improve software quality.

In certain scenarios, modifying design defects alter a high segment of the model radically or is occasionally equal to re-establishing a great part of the model. The refactoring solution that fixes every defect is not optimal because of the high code modification/adaptation needed. Suppose the suggested refactoring is beneficial to fix predicted defects with small code modifications by computing code change scores. The code change score is described in Equation (7), and Figure 7 shows the code change score of the suggested model.

ii Automatic refactoring score

Automatic refactoring eliminates dead and redundant code without altering a system's functionality and transforms unstructured codes into well-structured codes and procedural codes into object-oriented codes. Refactoring is an iterative cycle of creating a lesser program conversion, testing it to guarantee the exactness, and creating another minor transformation with unit testing. If at the first point a test fails, the final minor change is uncompleted and repetitive differently. Figure 8 demonstrates the automatic refactoring score of the suggested DNNFF model.

iii Defect correlation ratio

This study validates the suggested refactoring operation to fix the design defect by computing the defect correction ratio (DCR) on benchmarks collected from open-source models. The defect correction ratio is provided in expression 6, which relates to supplementing the rate of the number of design defects afterward refactorings (predicted utilizing the bad smell prediction rule) over the overall number of defects predicted earlier refactorings. The proposed DNNFF model increases the defect correlation ratio related to other existing models, and Figure 9 shows the defect correlation ratio.

iv Refactoring precision ratio

Refactoring precision denotes the outcomes of the code decisions. This study uses two validation approaches, automatic and manual validations, to assess the effectiveness of the suggested refactoring. For the manual validations, this study requested candidates of potential operators

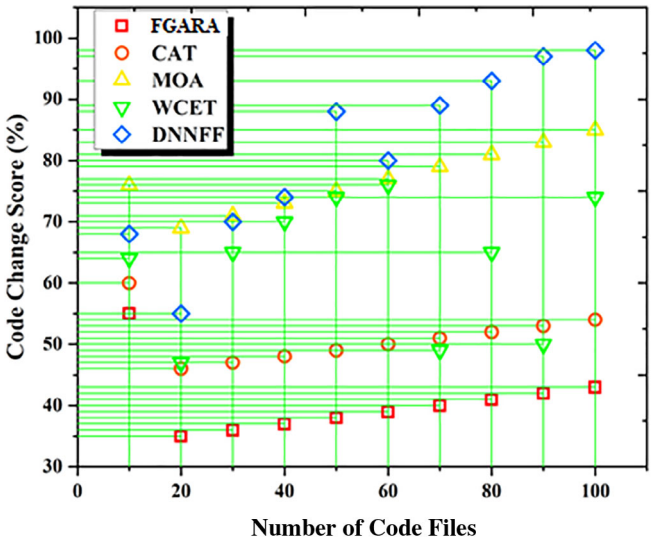

FIGURE 7 Code change score

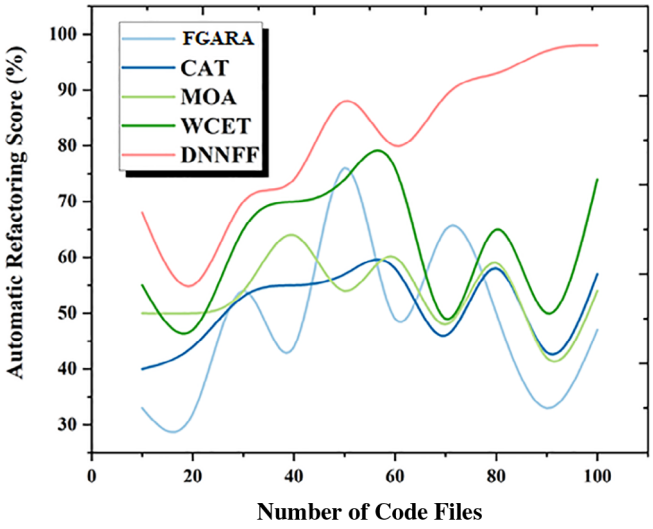

FIGURE 8 Automatic refactoring score

of our refactoring tools to assess whether the recommended refactoring is possible and makes sense semantically. This study defines the metric refactoring precision (RP), which resembles the number of significant refactoring operations (high-level and low-level), in terms of semantics, over the overall number of recommended refactoring operations. Refactoring precision is provided by

$$\mathrm{RP} = \frac{\text{Coherent Refactorings}}{\text{Suggested Refactorings}} \in [1, 0]. \tag{10}$$

This study compares the suggested refactorings with the predictable ones utilizing existing benchmarks for automatic validation. Figure 10 signifies the refactoring precision ratio.

v Execution time

It is significant to compare numerous execution outcomes with the execution period to assess our approach's performance and stability. The execution period for discovering the optimum refactoring solutions with several iterations was less than 48 min. Moreover, this study evaluates the influence of the number of recommended refactorings on the defect correlation ratio, refactoring precision, reused refactorings, and code change

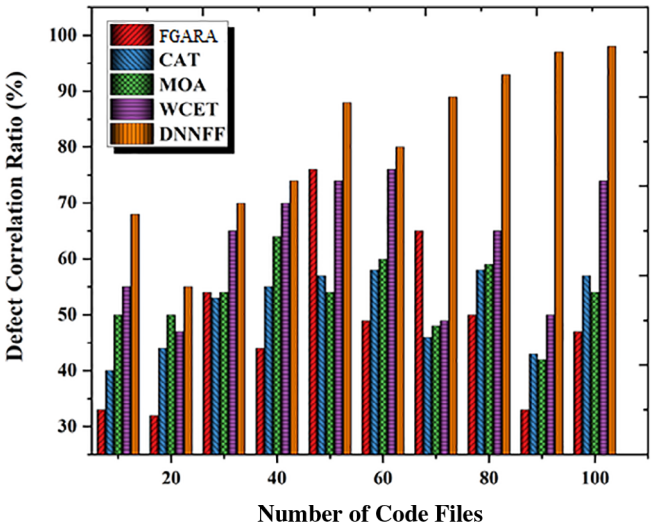

FIGURE 9 Defect correlation ratio

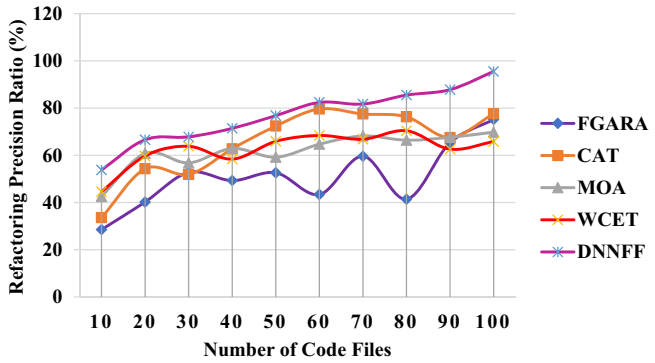

FIGURE 10 Refactoring precision ratio

scores in five different executions. However, as these variables are independent, they can be effortlessly constructed consistent with the developers' inclinations, for instance, if they want to decrease the execution period. Figure 11 shows the execution time.

Finding the best refactoring solutions across numerous iterations took less than 48 min.

vi Flaw detection ratio

This study proposed the DNNFF model to syndicate known approaches for result design flaws based on metrics with a deep learning process, such that design flaw detection is flexible to diverse views. When employing deep learning to intention flaw detection, measuring the performance compared to human intuition is necessary. This study depends on metrics to perceive design flaws. Adaption to the necessities of distinct users employing deep learning techniques has been added. A recent study states that design flaws in the object-oriented program can generate fraudulent code and propose automatic design flaw detections, using multi-pattern matching and detection rules reuse. Figure 12 demonstrates the flaw detection ratio of the suggested DNNFF model.

The results of various test parameters are depicted in a graphical form in the simulation analysis. There are comparisons between the proposed DNNFF model and other approaches in Figures 7–11. According to Figures 7–11, this new method outperforms the old approaches in Code Change Score, Automatic Refactoring Score, Defect Correlation Ratio, Refactoring Precision Ratio, execution time, and Flaw Detection Ratio.

To determine a DNNFF model which is most effective, on our dataset, in identifying the refactoring related commits we investigated the performance of four well-known ML algorithms (i.e., EPRA, CAT, MOA, and WCET). Note that, the evaluation of our approach in this study will base on such ML algorithms. The accuracy of identifying refactoring related commits is critical in deciding whether a feature request under implementation would demand refactoring or not. According to Figures 7–11 the code change score of predicting refactoring related commits ranges from 68.15%

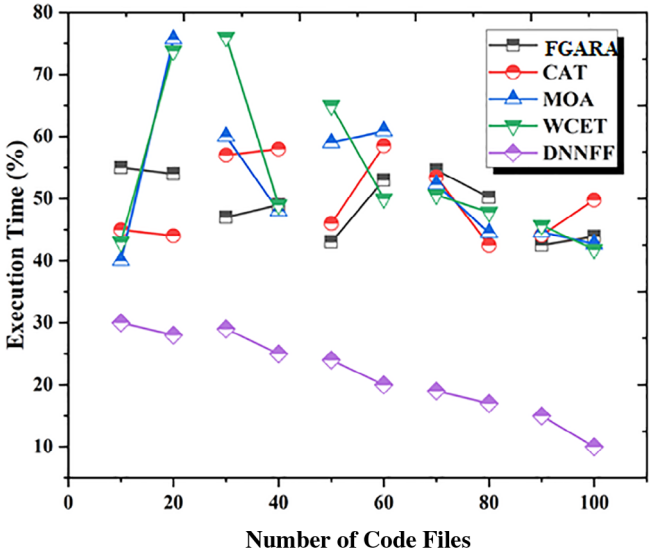

FIGURE 11 Execution time

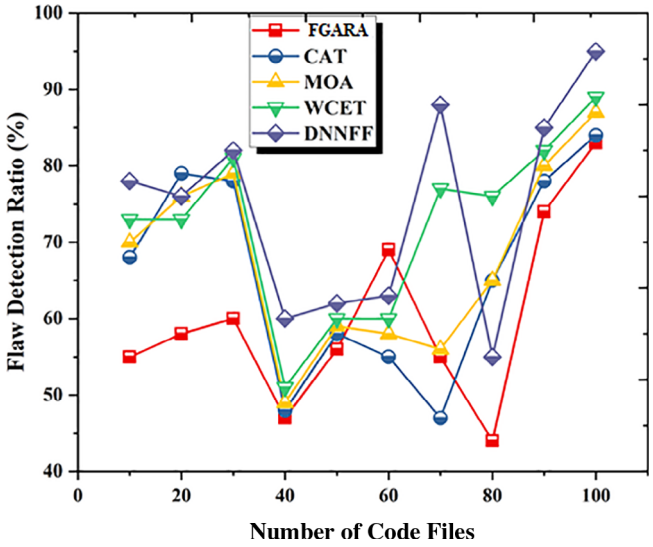

FIGURE 12 Flaw detection ratio

to 98.70%, whereas the Automatic Refactoring Score ranges from 67.22% to 97.30%. From the above diagram we observe that, the DNFF model appeared to be the best performing classifier on our dataset. It recorded a Defect Correlation Ratio, Refactoring Precision Ratio, Execution Time, and Flaw Detection Ratio ranges from 67.61% to 96.90%, 55.11% to 95.90%, 30.32% to 10.02%, and 98.53% to 94.40%, respectively. The suggested DNNFF model speeds up the execution time while increasing the code change score, automatic refactoring score, defect correlation ratio, refactoring precision ratio, and flaw detection ratio. In contrast to the Code Analysis Tool, the Many-Objective Approach, and the Worst-Case Execution Time, Fuzzy Genetic Automatic Refactoring (FGARA) is a preferred choice.

5 | CONCLUSION AND FUTURE WORKS

This article presented an interactive refactoring method that does not need a description of fitness functions. The designer is requested to assess manually refactoring resolutions recommended using Genetic Algorithms (GA) for minimum iterations; then, these instances are utilized as training sets for the DNNs to assess the solutions of the genetic algorithm in the succeeding iterations. This study evaluates our method for open-source models. This research reports the outcomes on the effectiveness and efficiency of our method compared to existing methodologies. A labeled

training dataset of Unified Modeling Language class diagrams metric values is utilized to infer a function that labels the figures with functional decompositions. Flaw detection is trailed by employing a sequence of the refactoring operation that discourse functional decomposition. The experimental results demonstrate that the suggested DNNFF model enhances the code change score of 98.7%, automatic refactoring score of 97.3%, defect correlation ratio of 96.9%, refactoring precision ratio of 95.9%, flaw detection ratio of 94.4%, and reduces the execution time of 10.2% compared to other existing methods. The outperformance is statistically significant, and the ranking is varied across different metrics. Compared to other well-known fuzzy genetic automatic refactoring techniques, the suggested DNNFF model excels in areas including code change score, automatic refactoring score, defect correlation ratio, and execution time. This study plans to examine empirical research to deliberate further structures and a higher set of refactoring operations in our experimentations in future work. This study plans to extend our method to reflect refactoring opportunities utilizing our interactive learning method.

CONFLICT OF INTEREST

The authors declare that there is no conflict of interest in regard to this research paper and the work quoted in this paper.

DATA AVAILABILITY STATEMENT

The data that support the findings of this study are openly available in repository name GHTorrent -Github at https://github.com/gousiosg/github-mirror.

ORCID

Chitti Babu Karakati https://orcid.org/0000-0001-6260-6858

REFERENCES

1. Baqais AAB, Alshayeb M. Automatic software refactoring: a systematic literature review. *Softw Qual J*. 2020;28(2):459-502.
2. Amudha G, Narayanasamy P. Distributed location and trust based replica detection in wireless sensor networks. *Wirel Pers Commun*. 2018;102(4):3303-3321.
3. Paixão M, Uchôa A, Bibiano AC, et al. Behind the intents: an in-depth empirical study on software refactoring in modern code review. In: Proceedings of the 17th International Conference on Mining Software Repositories; 2020:125-136.
4. Gunasekaran A, Narayanasamy P. Analysing the network performance of various replica detection algorithms in wireless sensor network. *J Comput Theor Nanosci*. 2018;15(3):989-994.
5. Janarthanan R, Doss S, Baskar S. Optimised unsupervised deep learning assisted reconstructed coder in the on-nodule wearable sensor for human activity recognition. *Measurement*. 2020;164:108050.
6. Lacerda G, Petrillo F, Pimenta M, Guéhéneuc YG. Code smells and refactoring: a tertiary systematic review of challenges and observations. *J Syst Softw*. 2020;167:110610.
7. Baskar S. Error recognition and correction enhanced decoding of hybrid codes for memory application. In: 2014 2nd International Conference on Devices, Circuits and Systems (ICDCS). IEEE; 2014:1-6.
8. Nguyen NT, Liu BH. The mobile sensor deployment problem and the target coverage problem in mobile wireless sensor networks are NP-hard. *IEEE Syst J*. 2018;13(2):1312-1315.
9. Kaur S, Awasthi LK, Sangal AL. A brief review on multi-objective software refactoring and a new method for its recommendation. *Arch Comput Methods Eng*. 2021;28(4):3087-3111.
10. Nguyen NT, Liu BH, Pham VT, Liou TY. An efficient minimum-latency collision-free scheduling algorithm for data aggregation in wireless sensor networks. *IEEE Syst J*. 2017;12(3):2214-2225.
11. Gao J, Wang H, Shen H. Smartly handling renewable energy instability in supporting a cloud datacenter. In: 2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS). IEEE; 2020:769-778.
12. Sheneamer AM. An automatic advisor for refactoring software clones based on machine learning. *IEEE Access*. 2020;8:124978-124988.
13. Shakeel PM, Baskar S, Fouad H, Manogaran G, Saravanan V, Montenegro-Marin CE. Internet of things forensic data analysis using machine learning to identify roots of data scavenging. *Fut Gen Comput Syst*. 2021;115:756-768.
14. Kaur A, Kaur M. Analysis of code refactoring impact on software quality: a scientific explanation. *Adv Aspects Eng Res*. 2021;7:43-52.
15. Khalifa NEM, Taha MHN, Manogaran G, Loey M. A deep learning model and machine learning methods for the classification of potential coronavirus treatments on a single human cell. *J Nanopart Res*. 2020;22(11):1-13.
16. Elhoseny M, Mohammed MA, Mostafa SA, et al. A new multi-agent feature wrapper machine learning approach for heart disease diagnosis. *Comput Mater Contin*. 2021;67:51-71.
17. Khalilipour A, Challenger M, Onat M, Gezgen H, Kardas G. Refactoring legacy software for layer separation. *Int J Softw Eng Knowl Eng*. 2021;31(2):217-247.
18. Gupta BB, Prajapati V, Nedjah N, Vijayakumar P, Abd El-Latif AA, Chang X. Machine learning and smart card based two-factor authentication scheme for preserving anonymity in telecare medical information system (TMIS). *Neural Comput Appl*. 2021;1-26. https://doi.org/10.1007/s00521-021-06152-x
19. Kaur S, Awasthi LK, Sangal AL. A review on software refactoring opportunity identification and sequencing in object-oriented software. *Rec Adv Electric Electron Eng*. 2021;14(3):252-267.
20. Manogaran G, Alazab M, Saravanan V, et al. Machine learning assisted information management scheme in service concentrated iot. *IEEE Trans Ind Inform*. 2020;17(4):2871-2879.
21. AlOmar EA, Mkaouer MW, Newman C, Ouni A. On preserving the behavior in software refactoring: a systematic mapping study. *Inform Softw Technol*. 2021;106675:106675.

22. Ullah F, Wang J, Jabbar S, Al-Turjman F, Alazab M. Source code authorship attribution using hybrid approach of program dependence graph and deep learning model. *IEEE Access*. 2019;7:141987-141999.
23. Alazab M. Profiling and classifying the behavior of malicious codes. *J Syst Softw*. 2015;100:91-102.
24. Nasagh RS, Shahidi M, Ashtiani M. A fuzzy genetic automatic refactoring approach to improve software maintainability and flexibility. *Soft Comput*. 2021;25(6):4295-4325.
25. Sharif KY. Code analysis tool to detect extract class refactoring activity in Vb. Net classes. *Turk J Comput Math Educ (TURCOMAT)*. 2021;12(3):2172-2177.
26. Mohan M, Greer D. Using a many-objective approach to investigate automated refactoring. *Inform Softw Technol*. 2019;112:83-101.
27. Meng F, Su X. WCET optimisation strategy based on source code refactoring. *Clust Comput*. 2019;22(3):5563-5572.
28. https://www.kaggle.com/datasets/aroojanwarkhan/fitness-data-trends.
29. Terra R, Valente MT, Miranda S, Sales V. Jmove: a novel heuristic and tool to detect move method refactoring opportunities. *J Syst Softw*. 2018;138:19-36.

How to cite this article: Karakati CB, Thirumaaran S. Software code refactoring based on deep neural network-based fitness function. *Concurrency Computat Pract Exper*. 2023;35(4):e7531. doi: 10.1002/cpe.7531