Received: October 23, 2021. Revised: December 14, 2021. 563

International Journal of
Intelligent Engineering & Systems

http://www.inass.org/

Extension of Deep Learning Based Feature Envy Detection for Misplaced Fields and Methods

Malathi Jeevanantham[1]* Jabez Jones[2]

[1]*Faculty of Computer Science and Engineering,*
Sathyabama Institute of Science and Technology, Chennai, Tamilnadu, India
[2]*Department of Information Technology,*
Sathyabama Institute of Science and Technology, Chennai, Tamilnadu, India
* Corresponding author's Email: mals_sakthiphd@yahoo.com

Abstract: Code smells (CS) are a severe symptom in software source code that leads to a serious problem in software maintenance and evolution. Feature Envy (FE) is a type of CS that refers to methods that are misplaced in the source code. Many tools like infusion, JDeodorant and JspIRIT have been used to identify the CS. The conversion of source code measurements into predictions is based on manually constructed heuristics. But, manually selecting good features and constructing optimal heuristics is a challenging issue. So, automatic detection of CS is required. The Deep Learning (DL) methods play an important role in achieving the automatic detection of FE(DLFED). The FE detection approach based on DL is intended to discover misplaced methods. The DL technique automatically identifies the source code characteristics for FE identification as well as automatically improves the difficult mapping structure between the features and their prediction. But, a DL-based automatic detection of CS requires a significant number of features with labeled training datasets to provide better prediction results. Therefore, an extension of the DL-based FE detection technique (EDLFED-FM) is proposed in this article to detect both misplaced methods and fields. The semantic relationship between identifiers in the source code program is used to extract the features of misplaced fields and methods. The new decomposition slice method is proposed to convert the extracted features of misplaced methods and fields into the slicing vector composition. This sliced feature helps deep learners understand the relationship between misplaced methods and fields. This automated method is capable of producing labeled training data for DL techniques-based classifiers without the requirement for human intervention. Finally, the experimental outcomes proved that precision rate of suggested method outperforms the JDeodorant and DLFED methods by 14.49% and 10.29 % for free plane, 14.72 % and 12.2 % for Junit, and 21.84% and 14.68 % for JExcelapi applications. Hence, it has been demonstrated that the predicted misplaced methods and fields are moved to the refactoring class for positive testing of source code program with less training data.

Keywords: Code smell, Feature envy, Deep learning, Program source code, Slicing vector composition.

1. Introduction

CS is a fundamental violation of software development principles, which will reduce the quality of the code. The presence of CS does not necessarily suggest that the software will not function properly; nonetheless, it can decelerate processing speed, raise the chance of failures and errors, and render the program susceptible to defects in the future. CS can lead to poor code quality and thereby it increases the technical debt [1].

CS denotes a more serious problem, but as the name implies, they can be detected or identified quickly. The best odour is one that is easy to detect but leads to a fascinating difficulty, such as classes with data but no behaviour. Some tools [2] like infusion, JDeodorant, PMD and JspIRIT may easily detect CS.

Since CS has many software characteristics with informal and subjective definitions, automatic detection of CS is important to achieve without any human need for refactoring operations in large

source code detection [3]. In recent times, Machine Learning (ML) based smell detection approaches have become an effective alternative because they do not only have the means of judging as humans but also, in the case of CS detection [4]. Furthermore, ML approaches may identify CS detection as smelly or non-smelly variant results. The issue emerges during the manual configuration phase, resulting in unbalance [5]. So, the DL techniques are the most effective method for detecting the CS in large source code programs.

The DL methods for automatic CS detection can be processed using some deep neural networks (DNN) like CNN and RNN, demonstrating the practical application of the DL model for detecting smells without the need for extensive feature engineering, which provides a source code in tokenized form. However, the DL struggles with detecting the text features and optimal heuristics in large source code programs. So, the FE CS is utilised in this DL approach for easy prediction of features and heuristics for the refactoring class with less training data [6].

In existing work, the FE detection technique based on DL is intended to detect only the misplaced methods in program code. Therefore, in this paper, the DL based FE detection is extended to find both misplaced methods and fields from source code program. The detection of both misplaced methods and fields of FE helps to eliminate deeper problem in the running software programs. This advanced DL technique will automatically choose source code features for FE detection while also improving the problematic mapping construction between the features and their prediction. This automated strategy will be capable of producing labelled training data for DL techniques-based classifiers without the use of human sources. The features are extracted by analyzing the semantically relationship between identifiers. Finally, the improved decomposition slice method is developed to transfer the extracted features into the slicing vector composition. This sliced feature assists deep learners in comprehending the connection between misplaced methods and fields. The predicted misplaced methods and fields are relocated to the refactoring class for positive testing of source code programs with less training data.

The rest of the manuscript is organized as follows: Deep learning based code smell detection techniques are explained in Section 2. The suggested EDLFED-FM is explained in Section 3, followed by a test experiment undertaken and reported in Section 4 and lastly, the study is concluded with its future work in Section 5.

2. Literature survey

A tool called WekaNose was introduced [7] to perform experiments using ML techniques to detect smells from v-code. This method intentionally sets the rules for obtaining the trained algorithms in order to categorize an instance (method or class) as affected or not by a CS. This tool helps to perform a DL method for identifying a certain type of CS. Then the correlations between CS and some specific metrics were developed. Finally, the rules for rule-based CS detectors were created to enhance the detection of CS. However, the rules could minimize the detection performance.

A Systematic Literature Review (SLR) employing ML techniques for CS Detection was created [8]. The four target CS prediction models were developed using this method (i) specific CS were considered, (ii) the ML method was used for the setup, (iii) different types of evaluation strategies were used, and (iv) the training strategies were used to train and evaluate the ML techniques. A meta-analysis of the ML models like decision trees and random forests was also conducted. However, the detection involves source code complexity.

The CS detection was proposed [9] using ML techniques like static code metrics and CS metrics in the dataset. Initially, two classification models like Bagging and Random Forest were used for enchaining the performance for CS detection. Then, the four approaches were developed. The first approach was identifying all twenty-seven characteristics in a data collection of CS. The second strategy evaluated the databases with three feature selection methods like $F_1, F_2,\ and\ F_3$ was employed for the datasets reduction. The third approach utilizes the ensemble learning method in conjunction with three aggregators as like first approach. The fourth approach considers the integrated dataset of the second stage with four combinations. However, memory and time constraints were high.

For detecting the CS [10] proposed a DL structure with two specific models. In this method, the DL structure was employed with two layers of CNN and RNN as well as Auto Encoders to modulate the performance to fine-tune the smell detection performances of different CS. The performance of the model was affected by adjusting the learning hyper parameters. The DL investigation on CS detection was transferred to analyze the trained model for detecting smells in a programming language, and it could also be used to detect on other languages. However, CS detection tools

utilized in this structure either not available or not matured.

A hybrid detection approach was presented [11] using unsupervised and supervised algorithms to detect four CSs. Initially, the dimensionality reduction approach was conducted using a deep auto-encoder for extracting the most significant features. The ANN classifier learns to generate the new data after the feature space has been decreased with a minor reconstruction error. This technique was tested using four datasets derived from a large number of open source systems. Furthermore, additional sorts of features were discovered using fine-grained data, and this included the detection of other forms of CS. However, this method does not generalise to industrial projects.

A method based on DL was devised [12] to detect two CS, such as Brain Class and Brain Method. To generate complex patterns in the higher layer, one of the DL architectures like the Convolution Neural Networks-1D (CNN) model was used in this method. The input was gathered from open source Java software, and the program code was then fed into a metrics analysis tool to obtain the metrics required to identify the brain class and brain function. Object-oriented metrics were assessed to detect the presence of smells in software. However, the approach necessitates a large number of datasets in order to detect CS.

A semantic-based technique was introduced [13] to detect code or bad smells in source code at various levels of granularity. The abstract syntax with variation Auto-encoder was created to distinguish the three CSs of blob, FE, and long method. Initially, the source code was parsed using an Abstract Syntax Tree, and the resulting trees were transformed into vector representations using a transformation. Then, the variation auto-encoder was employed to produce the deep representations that incorporate the required semantic features. Finally, the semantic features were fed into a logistic regression to identify whether there was a CS or not. However, this approach does not handle other forms of class-level smells.

A new metric was introduced [14] to detect FE CS, which may be eliminated by shifting the specific method. This suggested metric detects FE CS candidates using two methods, like similarity between methods and the distance between the method and the target class candidate. Then, the identified FE observed CS was removed using the removal technique. Following that, the source code was rebuilt using the refactoring system without affecting the system's behaviour. The tool JCodeCanine was created as an eclipse plug-in to evaluate the performance of new metrics. However, the computational complexity was high.

A tool named FEED (Feature Envy Detector) was developed [15] for identifying FE CS detection based on data flow analysis. The data flow analysis was used to create the DEF (definition) and Usage (use) information in order to discover the variables in each code statement. The FEED was utilized in the data flow analysis to pinpoint a statement for the actual problem for the FE. Furthermore, the FEED assists novice programmers in avoiding FE smells on large datasets. However, the procedure for using this tool was slightly complicated.

A new method was proposed [16] to detect and identify the CS called "Anti-pattern Detection and Identification using Oblique Decision Tree Evolution (ADIODE). To discover the imbalance data in the CS, the ADIODE uses the input as a base of anti-pattern instances with a collection of Oblique Decision Trees (ODTs) utilizing an Evolutionary Algorithm (EA). This method was utilized for four primary features: (i) the use of oblique splitting hyper-planes to discover the imbalanced data, particularly small disjuncts. (ii) The data-driven discretization approach is used to avoid an empty or a normal sub-class that contains all occurrences. (iii) AUC was regarded as a fitness function that was used to evaluate both unbalanced and balanced data. However, an imbalanced binary classification problem occurs in the CS detection.

An automated approach was created [17] for detecting bad code or monitoring code quality during the development process. Initially, this tool implementation uses a set of code, programming design principles, synchronous threads, and cache representation. This solution was built using the JFly eclipse plug-in tool, which was linked to the IDE via multiple extension points. The users' preferences might be readily defined as a "bad smell" metrics threshold. The JFly tool's performance resulted in the detection of poor or CS. However, with low-spec systems, performance and outcomes cannot be assured.

CS detection tools like JDeodorant, inFusion, PMD, and JSpIRIT were investigated [18]. MobileMedia and Health Watcher were used as target systems to detect code smells such as God Class, God Method, and FE detection. Calculating the recall and precision of tools in recognizing the CSs was used to determine the accuracy. When all four CS detection tools were compared, JDeodorant recognize the majority of the correct entities with lower precision and a higher recall, which increased the validation effort to capture the majority of the affected entities. However, it was not possible to run

the tools during commit tasks, which resulted in low performance.

A novel DL-Based FE Detection (DLFED) was presented [19] to detect different sorts of CSs. In this method, DNN and advanced DL methods were utilised to automatically choose useful source code features for CS detection and develop a complex mapping between such features and binary conversion predictions. This method would automatically generate the labelled training data for CS detection. This approach may be used to detect CSs in four different categories: FE, long methods (LM), Large classes (LS) and misplaced classes (MC). It was first tested on open-source programs that had smells introduced into them, then on the original source code of open-source applications that had not been changed. However, the generated training data was of poor quality.

The above methods detects FE only based on misplaced methods, but not considered misplaced fields. The feature of fields can be utilized in DL to improve the effectiveness of FE detection. This paper proposes slicing vector to extract the features of fields. The following section describe how DL method detects FE by utilizing both misplaced methods and misplaced fields

3. Proposed methodology

In this section, the DL based FE CS is constructed to detect the misplaced methods and fields from the FE-CS of a particular program code. The enhanced slicing method is proposed in this method for better understanding of misplaced methods and fields. Also, the refactoring software leads to identify the misplaced methods and fields in the structured source code program.

3.1 Smell detection using DL approach

A DL based FE detection [19] utilized features extracted for misplaced methods. The same Deep learning structure is utilized in this paper. However, the various categories of features used in DL structure means, the performance can be improved. A vast amount of labeled training data also required to construct complex DL classifiers for detecting CSs. So the DL based FE detection is extended for utilizing features of both misplaced methods and fields at various locations of a program. The training data set is constructed from various applications which can be utilized for identifying CSs for any applications. A sampling and training processes are repeated n times, resulting in n classifiers. By training several classifiers and voting on the final decisions, the bootstrap aggregating technique [19] is utilized to increase the robustness and accuracy of the CS identification system.

3.2 DL-based FE detection for misplaced methods and fields

3.2.1. Feature extraction for training data

The training data for FE detection is derived from source applications. A dataset should be constructed within a certain class to detect misplaced methods and fields by examining the semantic connection between their identifiers, because meaningful identifiers can reveal the responsibilities and actions of the associated entities. In this method, the misplaced method m and field f are relocated from their enclosing class ec to a target (another) class tc using both structural and textual information as code metrics. After that select the features for FE CS. The input training data is a quintuple as

$$input(m,f) = \\ < name(m), name(ec), name(tc), dist(m,ec), \\ dist(m,tc); \\ name(f), name(ec), name(tc), dist(f,ec), \\ dist(f,tc); > \quad (1)$$

Where, $name\ (e)$ is an identifier for the method name and field name of software entity e, method under investigation m, field under examination f, ec enclosing class of m and f, and probable target class tc. $Dist(m,c)$ and $dist(f,c)$ are the distances given by class c between method m and field f. However, there is an automatic semantical relationship contained in method and class names, such as $name(m)$, $name(f)$, $name(ec)$, and $name\ (tc)$. The feature information gathered is coupled with the names of the methods and fields, as well as the names of its enclosing class and prospective target class. Usually, a method and field should be defined within the class that will handle the method and field's behavior.

3.2.2. Textual feature extraction

The semantics embedded system is completely exploited in natural languages for text feature extraction, employing certain modern technologies such as DL, which is more beneficial for extracting the text from training input for text feature extraction.

For text feature extraction, the semantics embedded system is fully exploited in natural languages to employ some advanced technologies like DL, which is more useful for extracting the text

from training input for text feature extraction. It is also difficult to quantify the relationship between textual and numerical characteristics as code metrics with handcrafted algorithms. The words in identifiers are converted into similar length numerical vectors before being fed into neural networks as natural language identifiers. The conversion is accomplished using the well-known word2vector function. Word2vector is a powerful method for learning high-quality distributed vector representations with precise syntactic and semantic word links. Word2vector is basically a neural network that predicts words that are adjacent to it, i.e., words that come before and after it. This network has been developed to modify words into numerical vectors using the hidden layer, which is a function of the training process.

A series of words is partitioned according to capital letters and underscores for a certain identifier, such as method and field name or class name, and each word is transformed into a fixed-length numeric vector: vector:

$$name(e) = < w_1, w_2, \ldots, w_k > \quad (2)$$

$$= < V(w_1), V(w_2), \ldots, V(w_k) > \quad (3)$$

The $name\ (e)$ identifies the software entity e, and $< w_1, w_2, \ldots, w_k >$ is termed as a word sequence. With $word2vector$, $V\ (w_i)$ transforms word (w_i) into a defined-length numerical vector. The length of the word sequence for each identifier is limited to five for simplifying the neural network function. For example, if the identification includes more than five terms, just the top five are extracted. If it comprises fewer than five words, a special character with a vector of zeros will be appended to the sequence.

3.2.3. Slice vector feature extraction

DL is utilized to capture the structural information of slice profiles using slicing vectors. This is a critical step in detecting CS. A slice profile's slicing vector is a point in space with uniform dimensions. This is comparable to Deckard's [20] approach's characteristic vector. A characteristic vector is a numerical representation of a subtree. The dimensions of a characteristic vector are determined by the total number of distinct types

of full binary trees necessary to approximate a certain tree. Deckard generates vectors by traversing

the parse tree backwards. Then, it is enlarged in a variety of ways because each slicing vector has a fixed size given by the number of slicing fields generated using srcClone [21]. So, there is no need to traverse a tree to generate these vectors, nor is it necessary to compute the values in the vectors' dimensions. Table 1 displays the Slicing Vectors for the Slice Profiles' techniques and field levels (SPs).

For a particular field slice profile, the slicing vector is represented as sv has the following dimensions:

$$sv(v) = \langle |Def\ |, |Use\ |, |Dfls|, |Ptrs|, |Cmeds| \rangle \quad (4)$$

Def- distinguish between fields (variable) that have the same name but have distinct scopes.

Use - list of lines in which a field is used

$Dfls$ - Fields that are data dependent on the slicing variable

$Ptrs$ – refers list of slicing variables

$Cmeds$ - a list of methods that have been called with the slicing variable

The size of a slicing field is shown by each dimension. The $|Def\ |$ dimension, for example, specifies the number of code lines on which an identifier is defined or redefined. If two code parts are smelled, their vectors will be highly similar. Intuitively, even if a smell modifies a small portion of the original copy, their slicing vectors will not alter significantly. This is the only required variable information that remains after this encoding stage. As a result, the code structure and other information (such as field names) would be lost.

Table 1. The slicing vectors for methods and fields levels for the slice profile

Methods	Fields	Slicing Vectors(svs)	Function-Level
$sumProd$	i	⟨2, 3, 2, 0, 0⟩	⟨6, 4, 2, 0, 1, 4⟩
	prod	⟨2, 1, 0, 0, 1⟩	
	sum	⟨2, 1, 0, 0, 1⟩	
	n	⟨1, 1, 0, 0, 0⟩	
$sumProd\_E$	i	⟨2, 3, 2, 0, 0⟩	⟨7,4,2,0,1,4⟩
	prod	⟨2, 1, 0, 0, 1⟩	
	sum	⟨2, 1, 0, 0, 1⟩	
	n	⟨1, 1, 0, 0, 0⟩	

At the field level, the sv for each field in procedure $sumProd$ is similar to the corresponding sv for the same fields in functions $sumProd\ E$. This is the only required variable information that remains after this encoding stage. As a result, the code structure and other information (such as field names) would be lost. At the field level, the sv for each field in procedure $sumProd$ is similar to the corresponding sv for the same fields in functions $sumProd\ E$.

Therefore, the decreased slice profiles are constructed into a collection of vectors. A comparison of n slicing vectors, where n denotes the number of fields. $SumProd$ and $sumProd\ E$ have slicing vectors of $\langle 6, 4, 2, 0, 1, 4\rangle$ and $\langle 7, 4, 2, 0, 1, 4\rangle$ respectively. Each approach has the same number of fields. Given a system dictionary, the single pass technique is used to generate vectors for its slice profiles. This algorithm will demonstrates how to construct vectors for the cut contour's three levels of granularity. $srcClone$ yields the collection G, which contains all svs of the system.

3.2.4. Algorithm for slicing vectors generation

Input: $\mathcal{S}_{\mathfrak{D}}$: system dictionary
Output: $\mathcal{G}$: slicing vectors set
/* $sv(v), sv(m), sv(f\ )$ generation */
1. begin
2. $\boldsymbol{\mathcal{F}} \leftarrow-$ Set of files in $\mathcal{S}_{\mathfrak{D}}$
3. $\boldsymbol{\mathcal{M}} \leftarrow-$ Set of methods for each file
4. $\boldsymbol{\mathcal{V}} \leftarrow-$ Obtain field set for each method
5. $for\ \forall sp(f\ ) \in \mathcal{F}\ do$
6. $for\ \forall sp(m) \in \mathcal{M}\ do$
7. $for\ \forall sp(v) \in \mathcal{V}\ do$
8. $sv(v) \leftarrow \langle |De\ f\ |, \ldots, |C\ meds|\rangle$
9. $sv(m) \leftarrow \langle |De\ f\ |, \ldots, |C\ meds|, |V\ |\rangle$
10. $sv(f\ ) \leftarrow \langle |De\ f\ |, \ldots, |C\ meds|, |\mathcal{V}\ |, |\mathcal{M}\ |\rangle$
11. $G \leftarrow G \cup (sv(f\ ), sv(m), sv(v))$

3.3 Fusioning of feature vector for training data

DL techniques are used to identify CS by providing the training datasets. At first, well-known and high-quality open-source applications are downloaded. Second, a labelled training sample is created for each method m and field f from such application which are illustrated below:

1. Move method refactoring has been used to train method m and field f to be transferable to other classes. The Eclipse $JDT\ APIs$ are used in the training as it considers parameters and fields accessible in m to be potential references to target classes, and such classes are referred to as prospective target classes (ptc) for m. If ptc is null, the method m may be relocated. It can also be applied to any of the ptc classes.
2. Assume that method m can be relocated to a set of classes denoted by $ptc = \{tc_1, tc_2, \ldots, tc_k\}$. If ptc is empty, the function is denied and moved on to the next. Otherwise, proceed to the following step to generate labelled training data.
3. A fifty-fifty chance of obtaining positive training data with FE or negative training data without FE is determined at random.
4. The negative data is generated in the following manner. First, a potential target class tc_i is chosen at random from ptc. Second, Find the method-field distance $dist(m, ec)$, $dist(m, tc\ i)$, $dist(f, ec)$, and $dist(f, tc_i,)$ where ec the enclosing class of is m and f. Third, create the negative item ($ngItem$) and add it to the training data set.

$$ngItem = < input, output > \quad (5)$$

$$input = < name(m), name(ec), name(tc_i), dist(m, ec), dist(m, tc_i); name(f), name(ec), name(tc_i), dist(f, ec), dist(f, tc_i), sv(f\ ), sv(m), sv(v)\ > \quad (6)$$

$$output = 0 \quad (7)$$

5. The following is the generation of a positive item. First, choose a probable target class tc_i at random fromptc. Second, using Eclipse $APIs$, relocate m and f from their enclosing class ec to tc_i. Third, a positive item with the input is formed.
6. The following is how the positive term is formed. First, choose a probable target class tc_i at random from ptc. Then, using the Eclipse API, to move m and f from the surrounding class ec to tc_i . Third, a positive item is generated using a positive input.

$$< name(m), name(ec), name(tc_i), dist(m, ec), dist(m, tc_i) \quad (8)$$

$$< name(f), name(ec), name(tc_i), dist(f, ec), dist(f, tc_i)\ sv(f\ ), sv(m), sv(v)) > \quad (9)$$

$$output = 1 \quad (10)$$

The distances are determined specifically once the method and field have been moved.

3.4 Fusioning the feature vector for testing data

As like fusioning the features vector for training data, the same procedure is followed for the testing data which is generated for each of the potential classes in feature vector. The method m and field f might be relocated to a collection of classes denoted by $ptc = \{tc_1, tc_2, \ldots, tc_k\}$. If ptc is empty, it means that the method m and field f were unable to be conveyed, or that the method was refused and the procedure proceeded to the next stage. Otherwise, move to the following step to generate labeled training data. With or without FE, randomly generated positive or negative testing results will be generated.

The following is how the negative testing data is constructed. The distance of method and field $dist(m, ec)$, $dist(m, tc_i)$, $dist(f, ec)$, and $dist(f, tc_i)$ are used to compute the random potential target class tc i from ptc, where ec is the enclosing class of m and f. Third, construct the negative item ($ngItem$) and add it to the testing data set. The distance of method and field $dist(m, ec), dist(m, tc_i), dist(f, ec$ and $dist(f, tc_i)$ are used to compute the random potential target class tc_i from ptc, where ec is the enclosing class of m and f. Third, construct the negative item ($ngItem$) and add it to the testing data set.

$$ngItem = < input, output > \quad (11)$$

$$input = < name(m), name(ec), name(tc_i), dist(m, ec), dist(m, tc_i); name(f), name(ec), name(tc_i), dist(f, ec), dist(f, tc_i), sv(f), sv(m), sv(v) > \quad (12)$$

$$output = 0 \ for \quad (13)$$

The following is how the positive term is formed. First, choose a probable target class tc_i at random from ptc. Second, use Eclipse $7APIs$ to move m and f from their parent class ec to tc_i. Third, a positive item is created whose input is

$$< name(m), name(ec), name(tc_i), dist(m, ec), dist(m, tc_i) \quad (14)$$

$$< name(f), name(ec), name(tc_i), dist(f, ec), dist(f, tc_i), sv(f), sv(m), sv(v) > \quad (15)$$

$$output = 1 \quad (16)$$

The distances are calculated specifically once the technique and field have been relocated. Where ec is the method m specifying the class and field f to be used as input to the learnt deep NN.

3.5 Classification of FE

The structure of a deep NN-based classifier for FE CS detection is provided for the classification process. The classifier input is divided into two types as textual and numerical. The textual input consists of a word sequence formed by concatenating the names of the method and field, the enclosing class, and the names of possible destination classes. It is next to process by an embedding layer, which converts the text description into a numerical vector. These numerical vectors are then fed into a CNN. After that, the text description is passed via an embedding layer, which turns it to a numerical vector. After that, the numerical vectors are inputted into a CNN.

For the two CNN layers, the following parameters are specified as filters = 128, kernel size = 1, and activation = $tanh$. CNNs can be utilised for a variety of tasks. First, major breakthroughs in CNN have recently been made, allowing CNN to be more successful in terms of improving ML capacity and flexibility. A strong CNN layer aids in learning the deep semantic association between IDs when it comes to finding them. Second, CNN is well-suited to parallel processing on new powerful GPUs, which greatly reduces training time. A flatten layer is applied to CNN output, which lowers the input to a one-dimensional vector.

The numerical inputs (m, ec), $dist(m, tc_i)$, and $dist(f, ec)$, dist(f,tc i) are fed directly into another CNN, and the output is routed to a flatten layer. This CNN is set up in the same manner as the one described in the preceding paragraph. This CNN is replaced by various types of neural network layers, however the replacement has no effect on performance.

The merge layer eventually combines the textual and numerical inputs after concatenating a list of inputs such as the previously mentioned textual and numerical inputs. The dense and output layers combine the textual and numerical inputs to provide a single prediction output, implying that the method m and field f should be moved to the target class tc. The loss function in this case is binary cross entropy.

The aforementioned classification technique is used to explore the following methods m and fields f to forecast whether it is smelly or not. First, its potential target classes are identified using Eclipse JDT as $ptc = tc_1, tc_2, \ldots\ldots tc_k$. If ptc is null, it

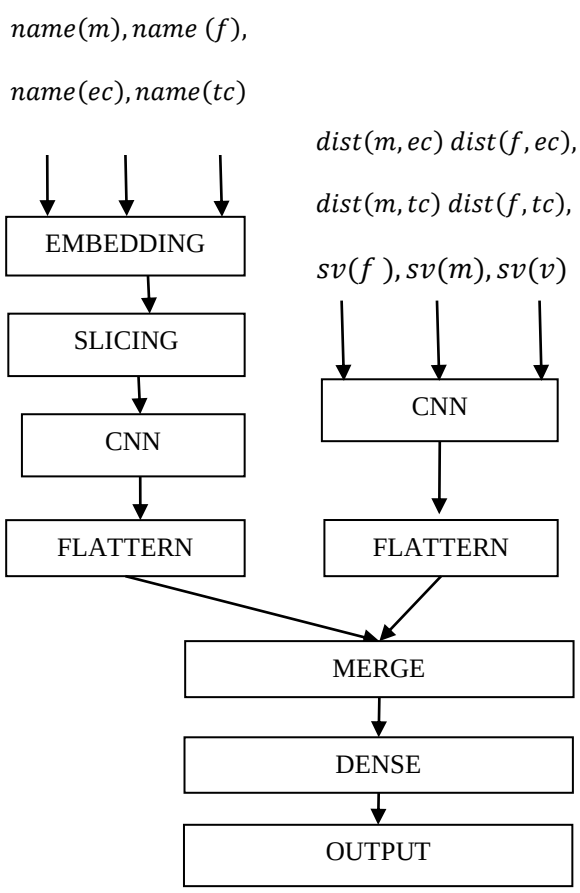

Figure. 1 The classifier detection for FE

signifies that the method and field could not be moved, which is not a good thing. If all of these data are anticipated to be negative (non-smelly), then the supplied method m and field f are unrelated to FE; otherwise, smelly is related to FE. Fig. 1 depicts the FE classifier detection.

3.6 Recommendation of refactoring solutions

The FE predicts that the misplaced methods m and fields f are stinky and that they should be moved using move method refactorings. If only one (noted as j) of the testing items prepared for m and f is expected to be positive, m and f should be assigned to the target class (tc_j)) associated with the positive testing data input j. If more than one testing data is expected to be positive, the method m and field f must be shifted to the class tc_i associated with the highest output (noted as input i). Despite the fact that the neural network is trained as a binary classifier with the goal of reducing misclassification rather than mean squared error, the neural network's output is a decimal range from 0 to 1. The neural network evaluates the prediction as affirmative if and only if the output exceeds certain thresholds.

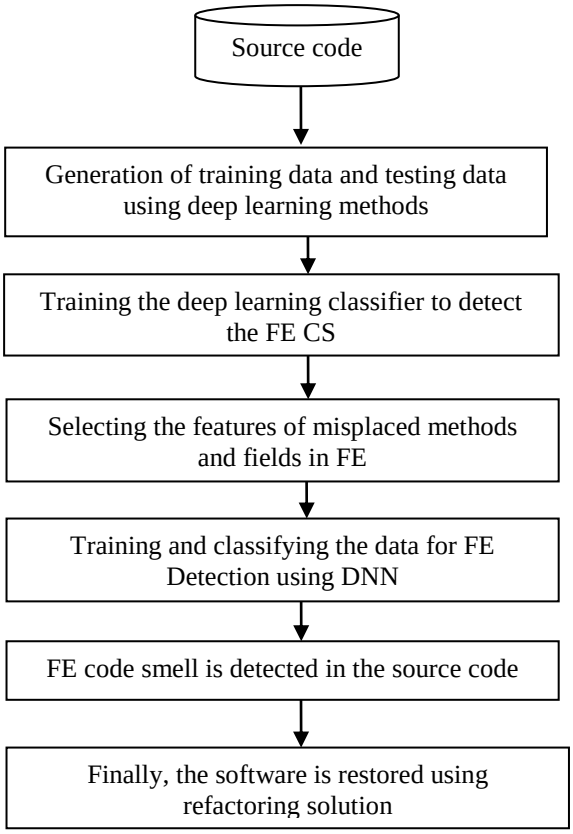

Figure. 2 Overall performances for detecting FE CS in the source code program

Fig. 2 represents the overall performance for detecting FE CS in the source code program.

4. Result and discussion

In this section, the performance of Extended DL Based FE Detection for Misplaced Fields and Methods (EDLFED-FM) is compared to that of existing FE approaches such as JDeodorant [18] and DL Based FE Detection (DLFED) [19], which is validated in terms of precision, recall, F1, MCC, and AUC using different applications such as Free plane Junit JExcelapi. For the experimental analysis, the training dataset for DL-based FE detection is compiled from well-known and high-quality open-source application. Then, each misplaced method m and field f is collected from such applications to produce the labeled training sample for better software refactoring entities to detect the FE CS. For the evaluation, the performance metrics and the comparison table for proposed and existing methods for defining the better performance in FE on different application like Free plane, Junit and JExcelapi. is defined as follows.

Table 2. Compared values of presented and existing methods on precision values for FE-CS detection

Applications	JDeodorant [18]	DLFED [19]	EDLFED-FM
Free plane	34.91	36.24	39.97
Junit	48.90	50.00	56.10
JExcelapi	32.04	34.04	39.04

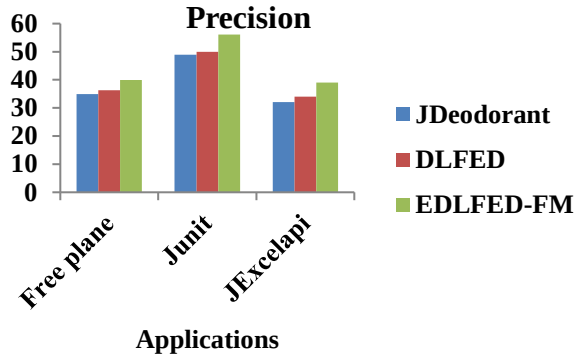

Figure. 3 Comparsion of presented and existing methods on precision values for FE-CS detection

Table 3. Compared values of presented and existing methods of recall values for FE-CS detection

Applications	JDeodorant [18]	DLFED [19]	EDLFED-FM
Free plane	12.63	94.14	95.21
Junit	24.53	82.22	87.43
JExcelapi	27.69	88.89	90.91

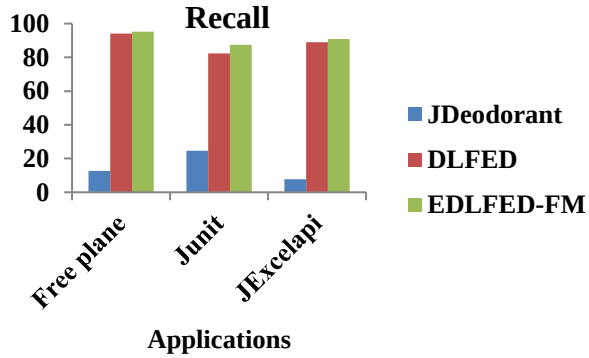

Figure. 4 Comparsion of presented and existing methods on recall values for FE-CS detection

4.1 Precision

The precision is used to determine a classifier's ability to forecast only the misplaced methods and fields from the FE CS in a given dataset. It is expressed as the percentage of accurately detected FE CS results at TP and FP rates, or the proportion of actual positives to detected CS.

$$Precision = \frac{True\ Positive(TP)}{True\ Positive(TP)+False\ Positive(FP)} \quad (17)$$

From the above Table 2 and Fig. 3, it is analyzed that the proposed EDLFED-FM is 14.49%, 10.29% for the given free plane, 14.72%, 12.2% for the given junit, 21.84%, 14.68% for the given JExcelapi applications which is resulted to be greater than that of existing JDeodorant and DLFED method. It is due to the DL model which augments the training data in difficult cases and highly focused on the misclassified code smells. Hence, it has been demonstrated that the proposed strategy will outperform all other existing methods for detection FE-CS in terms of precision.

4.2 Recall

Recall can be used to assess a model's ability to identify each of the feature vectors of interest in a set of data. It is expressed as the ratio of accurately detected CS results at TP and FN rates, or the proportion of observed positive CS.

$$Recall = \frac{True\ Positive(TP)}{Injected\ Smells} \quad (18)$$

From the above Table 3 and Fig. 4, it is analyzed that the proposed EDLFED-FM is 65.38%, 1.13% for the given free plane, 25.64%, 6.33% for the given junit, 10.82%, 2.27% for the given JExcelapi applications which is resulted to be greater than that of existing JDeodorant and DLFED method. The proposed EDLFED-FM provides the ratio of accurately classified FE-CS that emerged out to the total number of classified FE-CS. Hence, it has been demonstrated that the proposed strategy will outperform all other existing methods for detection FE-CS in terms of recall values.

4.3 F1-score

The approximate detection of FE CS is calculated using the harmonic average values of precision and recall.

$$F1 = 2 \times \frac{precision \times recall}{precision + recall} \quad (19)$$

From the above Table 4 and Fig. 5, it is analyzed that the proposed EDLFED-FM is 19.73%, 5.38% for the given free plane, 88.16%, 6.33% for the given Junit, 30.68%, 10.16% for the given JExcelapi applications which is resulted to be greater than that of existing JDeodorant and DLFED method. It is due to the slicing decomposition algorithm which eventually distributes the FE-CS to the training phase resulting in the exact FE-CS results. Hence, it has been demonstrated that the proposed strategy will outperform all other existing methods for detection FE-CS in terms of F1- score values.

Table 4. Compared values of presented and existing methods on F1-score values for FE-CS detection

Applications	JDeodorant [18]	DLFED [19]	EDLFED-FM
Free plane	18.55	52.33	55.15
Junit	35.14	62.18	66.12
JExcelapi	13.33	49.23	54.23

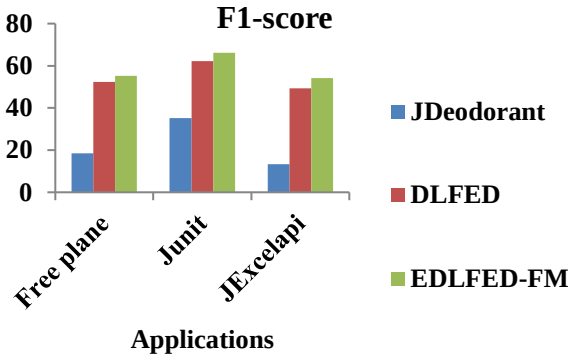

Figure. 5 Comparsion of presented and existing methods on F1-score values for FE-CS detection

4.4 Matthews correlation coefficient (MCC)

The MCC is a more dependable statistical rate that yields a high score only if the FE CS detection performed well in all four confusion matrix categories (TP, FN, TN, and FP), according to the number of positive and negative items in the dataset. The suggested destination for misplaced methods/fields is valid if and only if it promotes the method/fields to be relocated to its surrounding class/package before moving it. The MCC for CS detection is computed as

Table 5. Compared values of presented and existing methods on MCC values for FE-CS detection

Applications	JDeodorant [18]	DLFED [19]	EDLFED-FM
Free plane	7.41	36.69	39.61
Junit	27.55	46.02	49.02
JExcelapi	13.45	39.54	43.44

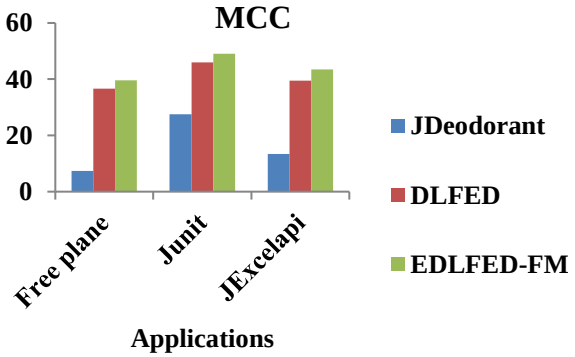

Figure. 6 Comparsion of presented and existing methods on MCC values for FE-CS detection

$$MCC = \frac{TP \times TN - FP \times FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}} \quad (20)$$

From the below Table 5 and Fig. 6, it is analyzed that the proposed EDLFED-FM is 43.45%, 7.96% for the given free plane, 77.93%, 6.52% for the given junit, 22.97%, 9.86% for the given JExcelapi applications which is resulted to be greater than that of existing JDeodorant and DLFED method. It is only due to the high resulted classification, if the prediction correctly classified a high percentage of negative code smell instances and a high percentage of positive code smell instances, with any class balance or imbalance. Hence, it has been demonstrated that the proposed strategy will outperform all other existing methods for detection FE-CS in terms of MCC values.

4.5 Area under curve (AUC)

The AUC is the measure of the ability of a classifier to distinguish between the misplaced methods and field from FE CS. The higher value of AUC provides the efficient detection of the FE CS as positive and negative results.

From the below Table 6 and Fig. 7, it is analyzed that the proposed EDLFED-FM is 78.77%, 4.71% for the given free plane, 97.13%, 4.32% for the given junit, 51.86%, 3.36% for the given JExcelapi applications which is resulted to be greater than that of existing JDeodorant and DLFED method. The above result has higher AUC which provides the better code smell classification system and

Table 6. Compared values of presented and existing methods on AUC values for FE-CS detection

Applications	JDeodorant [18]	DLFED [19]	EDLFED-FM
Free plane	48.67	83.10	87.01
Junit	45.36	85.72	89.42
JExcelapi	61.24	89.98	93.00

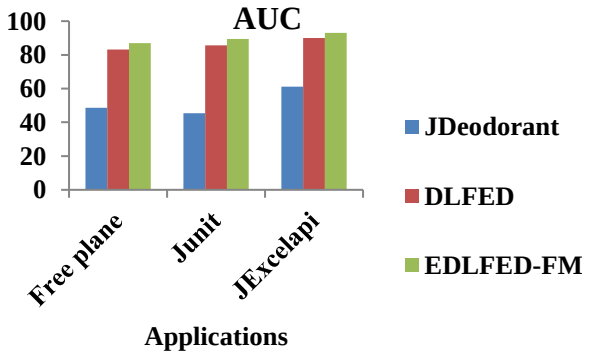

Figure. 7 Comparsion of presented and existing methods on AUC values for FE-CS detection

efficiently distinguishes the positive and negative classes in between. Hence, it has been demonstrated that the proposed strategy will outperform all other existing methods for detection FE-CS in terms of AUC values.

5. Conclusion

In an dynamic development environment, developers contribute by offering refactoring possibilities in the source code to reduce unwanted smells. The CS indicates that there are issues with the system's underlying structure (code), which may be fixed by restructuring the code (refactoring). The basis method used for removing the CS in program is manually supervising every lines of code in the software package. However, the manual inspection does not achieve the better result in removing the CS. This research work focuses on automatic detection of CS detection using DL technique for improving the quality of the software. By using the enhanced decomposition slice method the misplaced methods and fields are identified together in proposed FE-CS detection method. As a result, the proposed method EDLFED-FM achieves higher precision than existing methods JDeodorant and DLFED method that is 14.49%, 10.29% for free plane, 14.72%, 12.2% for junit, 21.84%, 14.68 % for JExcelapi applications. The higher precision rate leads to the program performing efficiently with less training data. In future, it will be interesting to apply the proposed approach to detect additional categories of code smells like large classes, misplaced classes, lazy class and data clumps.

Conflicts of Interest

The authors declare no conflict of interest.

Author Contributions

The paper conceptualization, methodology, software, validation, formal analysis, investigation, resources, data curation, writing—original draft preparation, writing—review and editing, visualization, have been done by 1st author. The supervision and project administration have been done by 2nd author.

References

[1] U. A. Mannan, I. Ahmed, R. A. M. Almurshed, D. Dig, and C. Jensen, "Understanding code smells in android applications", In: *Proc. of IEEE/ACM International Conference on Mobile Software Engineering and Systems*, pp. 225-236, 2016.

[2] X. Liu and C. Zhang, "The detection of code smell on software development: a mapping study", In: *Proc. of 5th International Conference on Machinery, Materials and Computing Technology, Atlantis Press*, pp. 5560-575, 2017.

[3] F. A. Fontana, P. Braione, and M. "Automatic detection of bad smells in code: an experimental assessment", *Journal of Object Technology*, Vol. 11, No. 2, pp. 5-11, 2012.

[4] D. D. Nucci, F. Palomba, D. A. Tamburri, A. Serebrenik, and A. D. Lucia, "Detecting code smells using machine learning techniques: are we there yet?", In: *Proc. of IEEE 25th International Conference on Software Analysis, Evolution and Reengineering*, pp. 612-621, 2018.

[5] G. Sibula and I. S. Gerr, "Finding software design disabilities with relational association rule mining", *Knowledge and Information Systems*, Vol. 42, No. 3, pp. 545-577, 2015.

[6] H. Liu, Z. Xu, and Y. Zou, "Deep learning based feature envy detection", In: *Proc. of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pp. 385-396, 2018.

[7] U. Azadi, F. A. Fontana, and M. Zanoni, "Poster: machine learning based code smell detection through WekaNose", In: *Proc. of IEEE/ACM 40th International Conference on Software Engineering: Companion Proceedings*, pp. 288-289, 2018.

[8] M. I. Azeem, F. Palomba, L. Shi, and Q. Wang, "Machine learning techniques for code smell detection: a systematic literature review and meta-analysis", *Information and Software Technology*, Vol. 108, pp. 115-138, 2019.

[9] I. Kaur and A. Kaur, "A novel four-way approach designed with ensemble feature selection for code smell detection", *IEEE Access*, Vol. 9, pp. 8695-8707, 2021.

[10] T. Sharma, V. Efstathiou, P. Louridas, and D. Spinellis, "Code smell detection by deep direct-learning and transfer-learning", *Journal of Systems and Software*, Vol. 176, p. 110936, 2021.

[11] M. H. Kacem and N. Bouassida, "A hybrid approach to detect code smells using deep learning", In: *Proc. of ENASE*, pp. 137-146, 2018.

[12] A. K. Das, S. Yadav, and S. Dhal, "Detecting code smells using deep learning", In: *Proc. of TENCON IEEE Region 10 Conference*, pp. 2081-2086, 2019.

[13] M. H. Kacem and N. Bouassida, "Deep representation learning for code smells detection using variational auto-encoder", In: *Proc. of International Joint Conference on Neural Networks*, pp. 1-8, 2019.

[14] K. Nongpong, "Feature envy factor: a metric for automatic feature envy detection", In: *Proc. of IEEE 7th International Conference on Knowledge and Smart Technology*, pp. 7-12, 2015.

[15] W. K. Chen, C. H. Liu, and B. H. Li, "A feature envy detection method based on dataflow analysis", In: *Proc. of IEEE 42nd Annual Computer Software and Applications Conference*, Vol. 2, pp. 14-19, 2018.

[16] S. Boutaib, S. Bechikh, F. Palomba, M. Elarbi, Makhlouf, and L. B. Said, "Code smell detection and identification in imbalanced environments", *Expert Systems with Applications*, Vol. 166, p. 114076, 2021.

[17] M. Hammad and A. Labadi, "Automatic detection of bad smells from code changes", *International Review on Computers and Software*, Vol. 11, pp. 1016-1027, 2016.

[18] T. Paiva, A. Damasceno, E. Figueiredo, and C. S. Anna, "On the evaluation of code smells and detection tools", *Journal of Software Engineering Research and Development*, Vol. 5, No. 1, pp. 1-28, 2017.

[19] H. Liu, J. Jin, Z. Xu, Y. Bu, Y. Zou, and L. Zhang, "Deep learning based code smell detection", *IEEE Transactions on Software Engineering*, 2019.

[20] H. W. Alomari and M. Stephan, "srcClone: detecting code clones via decompositional slicing", In: *Proc. of the 28th International Conference on Program Comprehension*, pp. 274-284, 2020.

[21] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard: scalable and accurate tree-based detection of code clones", In: *Proc. of IEEE 29th International Conference on Software Engineering*, pp. 96-105, 2007.