This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2019.2936376, IEEE Transactions on Software Engineering

Deep Learning Based Code Smell Detection

Hui Liu, Jiahao Jin, Zhifeng Xu, Yanzhen Zou, Yifan Bu and Lu Zhang

Abstract—Code smells are structures in the source code that suggest the possibility of refactorings. Consequently, developers may identify refactoring opportunities by detecting code smells. However, manual identification of code smells is challenging and tedious. To this end, a number of approaches have been proposed to identify code smells automatically or semi-automatically. Most of such approaches rely on manually designed heuristics to map manually selected source code metrics into predictions. However, it is challenging to manually select the best features. It is also difficult to manually construct the optimal heuristics. To this end, in this paper we propose a deep learning based novel approach to detecting code smells. The key insight is that deep neural networks and advanced deep learning techniques could automatically select features of source code for code smell detection, and could automatically build the complex mapping between such features and predictions. A big challenge for deep learning based smell detection is that deep learning often requires a large number of labeled training data (to tune a large number of parameters within the employed deep neural network) whereas existing datasets for code smell detection are rather small. To this end, we propose an automatic approach to generating labeled training data for the neural network based classifier, which does not require any human intervention. As an initial try, we apply the proposed approach to four common and well-known code smells, i.e., feature envy, long method, large class, and misplaced class. Evaluation results on open-source applications suggest that the proposed approach significantly improves the state-of-the-art.

Index Terms—Software Refactoring, Code Smells, Identification, Deep Learning, Quality

1 Introduction

Software refactoring is an effective means to improve software quality. It restructures the internal structures of software applications while preserving their external behaviors [1], [2]. Software refactoring has been well-supported by most of the modern IDEs, e.g., *Eclipse*, *Visual Studio*, and *IntelliJ IDEA*. For example, *Eclipse* has a top-level menu specially designed for software refactoring. The menu provides entries to most of the popular software refactorings investigated by the research community [3], [4], [5].

A key step in software refactoring is to identify where refactorings should be applied [1]. To facilitate the identification, Beck and Fowler [6] propose the concept of *code smells* that are '*certain structures in the code that suggest (sometimes they scream for) the possibility of refactoring*'. They introduce 22 types of code smells, including the well-known *feature envy* and *long method*. They also analyze their features, as well as their impact and potential solutions (refactorings). As a result, developers can identify when, where, and how to refactor by detecting code smells in software applications. Notably, besides code smells, there are other motivations for software refactorings [7], and some of them are even more prevalent than code smells. For example, the experiment conducted by Silva et al. [7] suggests that software refactorings are driven more often by changes in requirements than by smells in source code.

However, it is tedious and time consuming to manually identify code smells [1], [8]. Consequently, a large number of automatic or semi-automatic approaches have been proposed to detect different kinds of code smells [9], [10], [1]. Some of such approaches are introduced in Section 2. More comprehensive literature review and analysis on code smell detection have been made by Zhang et al. [10], Dallal [11], and more recently by Sharma and Spinellis [12].

Most of the existing code smell detection approaches rely on manually designed heuristics to map manually defined/selected code metrics into binary predictions, i.e., smelly or non-smelly [10], [11], [13]. However, it is challenging to manually select the best features. It is also difficult to manually construct the optimal heuristics. Analysis results on such approaches [10], [11], [14] also suggest that different people may select different metrics and different heuristics for the same code smells, which results in low agreement between different detectors [15]. To avoid manually designed heuristics, statistical machine learning techniques, like SVM, Naive Bayes, and LDA, are employed to build the complex mapping between code metrics (as well as lexical similarity) and predictions [16], [17]. However, empirical studies [18] suggest that such statistical machine learning based smell detection approaches have critical limitations that deserve further research.

To this end, in this paper we propose a deep learning based approach to detecting code smells. The key insight is that deep neural networks and advanced deep learning techniques could automatically select useful features from source code for code smell detection, and build the complex mapping between such features and the labels (smelly or not). Deep neural networks and advanced deep learning techniques have been proved good at selecting useful features and building complex mapping from input to output automatically [19]. With significant advances in deep learning techniques, they have been successfully used in different domains, e.g., natural language processing (NLP) [20],

- *H. Liu, J.H. Jin, Z.F. Xu, and Y.F. Bu are with the School of Computer Science and Technology, Beijinag Institute of Technology, Beijing 100081, China. E-mail: liuhui08@bit.edu.cn, jinjiahao1993@gmail.com, 848602422@qq.com, yifan_bu@qq.com.*
- *Y.Z. Zou and L. Zhang are with the Key Laboratory of High Confidence Software Technologies, Ministry of Education (Peking University), Beijing 100871. E-mail: zouyz@pku.edu.cn, zhanglu@sei.pku.edu.cn.*

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2019.2936376, IEEE Transactions on Software Engineering

video processing [21], speech recognition [22], and software engineering [23]. That is the reason why we employ deep learning techniques in this paper to build a neural network based classifier that classifies software entities in subject applications into '*smelly*' and '*non-smelly*'.

How to collect large training data is one of the biggest challenges for deep learning based code smell detection. To train the neural network based classifier, we need a large number of labeled training data to tune a large number of parameters contained in the deep neural network. However, existing datasets for code smell detection are rather small. To collect labeled training data for smell detection, existing approaches often rely heavily on manual checking of the initial detection results of similar smell detectors [16], [14]. However, manual checking is time consuming, which significantly limits the size of labeled training data, and thus prevents machine learning techniques (especially deep learning techniques) from reaching their maximal potential. To this end, in this paper we propose an automatic approach to generating labeled training data automatically without any human intervention. We generate negative samples, i.e., non-smelly software entities, by extracting software entities (and their context) directly from high quality open-source applications, assuming that such entities are well-designed. To generate positive samples, i.e., smelly software entities, we create code smells automatically by applying refactorings on well-designed source code. For example, we create feature envy smells by moving methods randomly to other classes where it could be moved by *move method* refactoring. After the movement, such methods (together with their new context) are taken as smelly methods (with *feature envy*). The refactorings that we employ to create code smells are different from traditional one, and thus we call them *smell-introducing refactorings*. Traditional refactorings are applied to smelly software entities to improve software quality whereas the smell-introducing refactorings are applied to well-designed entities to create code smells. However, smell-introducing refactorings, as well as traditional refactorings, preserve the external behaviors of software applications and could be conducted automatically by IDEs. The proposed approach can generate a large number of labeled training data for smell detection because the generation is fully automatic and no human intervention is required. The large data in turn serve as the basis for deep learning based smell detection.

The proposed approach is evaluated on four common code smells [6], i.e., *feature envy*, *long method*, *large class*, and *misplaced class*. Feature envy refers to such methods that are '*more interested in a class other than the one it actually is in*'. Long methods refer to lengthy methods that should be decomposed into multiple shorter ones to improve software readability and reusability. Notably, long methods are not equal to methods with great LOC: lengthy but simple and cohesive methods rarely deserve decomposition. Large classes refer to lengthy and low-cohesion classes that should be decomposed into multiple classes. Misplaced classes are those that are improperly distributed, and thus should be moved to the packages where it belongs. The evaluation is composed of two parts. In the first part, we evaluate the proposed approach on ten well-known open-source applications with automatically injected code smells. Evaluation results suggest that the proposed approach significantly outperforms existing approaches. It improves F-measure by 27.4% in feature envy detection, 15.11% in long method detection, 4.73% in large class detection, and 48.18% in misplaced class detection. In the second part, we evaluate the proposed approach on five open-source applications, and three independent developers manually validate the detected instances to classify them as true and false positives. Evaluation results also suggest that the proposed approach significantly improves the state-of-the-art.

The paper is an expanded version of our confer ence paper [24] that was published recently. Compared to the conference version, this paper makes the following expansions:

- It generalizes the approach proposed in the conference version [24]. The conference version is confined to a specific category of code smells (i.e., feature envy), and thus it cannot be applied directly to other code smells. In this paper, we propose a generic approach and successfully apply it to multiple categories of code smells (i.e., feature envy, long method, large class, and misplaced class).
- It improves the neural network based classification algorithm with bootstrap aggregating [25], [26]. It generate several bootstrap samples simultaneously from a given training dataset, and train multiple binary classifiers that in turn determine the final classification by voting. Evaluation results suggest that the new algorithm improves the classification performance.

The rest of the paper is structured as follows. Section 2 introduces related research. Section 3 proposes the generic approach for deep learning based code smell detection. Section 4 applies the generic approach to feature envy detection whereas Section 5 applies it to long method detection. Section 6 and Section 7 apply the proposed approach to the detection of large classes and misplaced classes, respectively. Section 8 and Section 9 present the evaluation of the proposed approach. Section 10 discusses related issues. Section 11 makes conclusions.

2 Related Work

2.1 Machine Learning Based Smell Detection

With the advance in machine learning techniques, a number of machine learning based smell detection approaches are proposed [16]. Kreimer [27] proposes a decision tree based approach to identify code smells, e.g, *long method* and *large class*. Vaucher et al. [28], [29], [30] apply Bayesian beliefs networks to detect *God class* (*Blob class*). Maiga et al. [31], [32] exploit Support Vector Machine (SVM) in detection of *Blob class*, and the same technology is employed by Amorim et al. [33] to detect *Blob class*, *long parameter list*, *long method*, and *feature envy*. Fontana et al. [34], [35], [17] compare different machine learning techniques (including J48, JRip, ERandom Forest, Baive Bayes, SMO, and LibSVM) in predicting the severity of code smells, e.g., *God class*, *data class*, *long method*, and *feature envy*.

Such machine learning based approaches have proved to be effective and efficient although some experimental evaluation also reveals their significant limitations [18].

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2019.2936376, IEEE Transactions on Software Engineering

The proposed approach differs from such approaches in the flowing aspects. First, the proposed approach exploits deep learning whereas they exploit traditional statistical machine learning techniques. Second, the proposed approach generates training data automatically whereas they collect training data manually with the help of smell detection tools.

2.2 Detection of Feature Envy

Beck and Fowler [6] propose the concept of *feature envy* to indicate such methods that are '*more interested in a class other than the one it actually is in*'. To improve software quality and to easy software maintenance, such misplaced methods should be moved to classes (by move method refactoring) that they are really interested in.

A number of approaches have been proposed to identify feature envy or move method opportunities [36]. The first one was proposed by Simon et al. in 2001 [37]. They define a distance to measure how closely two entities are related:

$$distance(e_1, e_2) = 1 - \frac{|p(e_1) \cap p(e_2)|}{|p(e_1) \cup p(e_2)|} \quad (1)$$

where e_1 and e_2 are two software entities, and $p(e)$ is the set of properties that are possessed by e. If e is a method, $p(e)$ includes e itself, all methods that are directly invoked by e, and all attributes that are directly accessed by e. If e is an attribute, $p(e)$ includes e itself, and all methods that directly access e. Base on the distance metrics, Simon et al. draw entities on a graph, and the geometric distances between entities correspond to the distance calculated by Formula 1. If a method is closer to entities of another class than those of its enclosing class, it is associated with feature envy.

Seng et al. [38] propose a search based approach to identify move method opportunities. They define a fitness function:

$$fitness(s) = \sum_{i=1}^{n} w_i * \frac{M_i(s) - M_{init_i}(s)}{M_{max_i}(s) - M_{init_i}(s)} \quad (2)$$

where s is the application to be refactored, $M(s)$ is a vector composed of seven metrics: weighted method count, response for class, information-flow-based coupling, tight class cohesion, information-flow-base-cohesion, lack of cohesion, and stability. $M_{init_i}(s)$ is the initial value of the metrics, and $M_{max_i}(s)$ is the maximal values obtained by a calibration run optimizing each metric alone beforehand. They employ a search algorithm to find out the optimal class structure, as well as a sequence of move method refactorings that turn the current system into the optimal one. This approach is the first search-based approach to identifying move method refactoring opportunities.

Tsantalis and Chatzigeorgiou [39] propose a metrics based approach to identify feature envy and move method opportunities. For each entity e (attribute or method), they collect a set of the entities (noted as S_e) that it accesses (if it is a method) or it is accessed from (if it is an attribute). They also define the distance between method m and class C. If m does not belong to C, the distance is computed as follows:

$$distance(m, C) = 1 - \frac{S_m \cap S_C}{S_m \cup S_C},\ where\ S_C = \bigcup_{e_i \in C} \{e_i\} \quad (3)$$

Otherwise, the distance is computed as follows:

$$distance(m, C) = 1 - \frac{S_m \cap S'_C}{S_m \cup S'_C},\ where\ S'_C = S_C \backslash \{m\} \quad (4)$$

Based on this distance metrics, they suggest to move method e to class C_{target} if 1) C_{target} has the shortest distance to e and 2) the movement satisfies some preconditions that ensure the movement will not change the external behaviors of the involved application. The distance is different from that defined by Simon et al. [37]. The latter measures the distance between two methods (or attributes) whereas the former measures the distance between a method and a class. This approach has been implemented by *JDeodorant*, a well-known, powerful, and open-source code smell detection tool. *JDeodorant* is the most commonly used benchmark in code smell detection research community.

Sales et al.[40], [41] propose a dependency based approach (called *JMove*) to identify feature envy. They define a metrics to measure the similarity of the dependencies established by a source method with the dependencies established by the methods in possible target classes. Based on the metrics, they represent the similarity between method m and class c as the average similarity between m and methods in c. They suggest to move method m to class c if c has the greatest similarity with m. Their evaluation results suggest that *JMove* is highly accurate and it outperforms the well-known *JDeodorant* in identifying feature envy.

Bavota et al. [36] exploit textual information in feature envy detection. They recommend move method opportunities via Relational Topic Models (RTM), a statistical mode. This model is employed to compute the relationship among methods according to structural similarity between methods as well as textual information (e.g., identifier names and comments) extracted from the source code. To the best of our knowledge, they are the first to identify move method opportunities based on textual information besides source code structures. Palomba et al. [42] also exploit textual information to detect code smells, including feature envy. If a method is lexically more similar to another class than its enclosing class, they suggest that the method is associated with feature envy.

Palomba et al. [43] propose a change based approach to identify feature envy. They assume that a method affected by feature envy changes more often with the envied class than with the class where it is defined. Consequently, if a method m is involved in commits with methods of another class (C_{target}) significantly more often than methods of its enclosing class, they suggests to move m to C_{target}. They are the first to identify feature envy by mining version histories of source code.

Liu et al. [44] propose a novel approach to recommend move method opportunities based on conducted refactorings. Once a method m is moved from class C_{source} to another class C_{target}, the approach checks other methods within C_{source}, and suggests to move the method who has the greatest similarity and strongest relationship with m. The rational is that similar and closely related methods should be moved together. They are the first to identify move method opportunities based on refactoring history.

Notably, most of such approaches and tools (e.g., *JDeodorant* and *JMove*) are designed to recommend refactor-

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2019.2936376, IEEE Transactions on Software Engineering

ings instead of detecting code smells. They find behavior-preserving refactoring opportunities by examining a list of preconditions. Consequently, some methods suffering from feature envy may not be reported, because there is no behavior-preserving refactoring that can be applied on them.

The proposed approach differs from such approaches in that it exploits deep learning techniques and generates training data automatically. The proposed approach is the first one to detect feature envy with deep learning techniques.

2.3 Identification of Long Methods

Code metrics are widely employed to detect long methods as well as other code smells [45], [46], [47]. Marinescu [45] analyzes the distribution of method complexity (McCabe's cyclomatic complexity), and reports the most complex (e.g., top 25%) methods as long methods. Evaluation results suggest that this simple approach works well and the precision varies from 50% to 75%. Lanza and Marinescu [46] detect long methods based on a set of code metrics including Lines Of Code (LOC), McCabe's Cyclomatic Complexity (CYCLO), Maximal Nesting Level (MAXNESTING), and Number of Accessed Variables (NOAV). If all such metrics of a given method are greater than their predefined thresholds (i.e., Formula 5 holds), the method is reported as a long method and is suggested to be decomposed.

$$(LOC > \ alpha) \wedge (CYCLO > \beta) \\ \wedge (MAXNESTING > \gamma) \wedge (NOAV > \delta) \quad (5)$$

DECOR proposed by Moha et al. [48] identifies long methods with LOC only:

$$LOC\_METHOD \geq very\_high \quad (6)$$

Yoshida et al. [49] propose a cohesion based approach to identify long methods, and their evaluation results suggest that their approach is accurate. Charalampidou et al. [47] empirically investigate the ability of size and cohesion metrics in identification of long methods. Evaluation results suggest that size metrics (i.e.,LOC) and cohesion metrics are capable of detecting and ranking long method bad smells. Similar empirical study is also reported by Charalampidou et al. [50], which confirms that code metrics could be employed to identify long methods.

Another way to detect long methods is to identify code fragments (or slices) that could and should be extracted from their enclosing methods. The enclosing methods of such fragments (or slices) are reported as long methods and are suggested to be refactored by *extract method* refactoring. A typical example is *JDeodorant* [51], [52], a well-known and widely used refactoring tool. It automatically detects long methods as well as other code smells. With program dependence graphs [53] and block-based slicing techniques [54], *JDeodorant* tries to identify source code slices from methods that could be extracted as a new method. A code slice is extractable if it adheres to all of the following three principles:

1) The code slice should contain the complete computation of a given variable declared in the original method;
2) The behavior of the program should be preserved after the application of the *extract method* refactoring;
3) The code slice should not be excessively duplicated in the original method.

Whenever *JDeodorant* identifies an extractable code slice from a method, it reports the method as a long method, and suggests developers to extract the code slice as a new method.

Similar to *JDeodorant*, *JExtract* proposed by Danilo Silva et al. [55] also identifies extractable fragments from methods. To generate candidate fragments, *JExtract* relies on a hierarchical model to represent the block structure of the source code within a method. From each of the blocks, *JExtract* generates automatically a list of fragments (i.e., subsequences of continuous statements within the block). If any of the generated fragments is extractable, i.e., satisfying syntactical, behavior-preservation, and quality preconditions, *JExtract* suggests to restructure the enclosing method by extracting the extractable fragment as a new method.

SEMI proposed by Charalampidou et al. [56] identifies extract method refactoring opportunities by checking methods' conformance to Single Responsibility Principle (SRP). In the first step, it recognizes fragments within a give method that collaborate together to provide a single functionality. Second, if extracting such fragments out as a separated method could reduce the diverse functionalities of the enclosing method, *SEMI* would take it as a refactoring candidate. *SEMI* groups and ranks such candidates, and recommends the one that would result in the greatest benefits.

GEMS proposed by Xu et al. [57] recommends extract method refactoring opportunities for a given method. It employs both structural and functional features of candidate refactorings, and makes recommendations based on a probabilistic model trained in advance. To the best of our knowledge, it is the first learning-based approach to recommending extract method refactoring opportunities. It leverages decision trees and gradient boosting to compute the probability of candidate refactorings. To collect training data, *GEMS* generates positive training data by applying *inline method* refactorings, and generates negative data by randomly selecting candidates (i.e., code fragments) from methods where *extract method* refactorings are required. The proposed approach is similar to *GEMS* in that both of them are learning based and both of them generate training data automatically. The proposed approach differs from *GEMS* in the following aspects. First, the proposed approach identifies long methods (code smells) whereas *GEMS* identifies which part of a given method should be extracted (i.e., extract method refactoring opportunities). As a result, it is challenge, if not impossible, to quantitatively compare their performance. Second, the proposed approach employs deep learning techniques whereas *GEMS* leverages decision trees.

All such approaches have greatly advanced the long method detection and extract method refactoring. The proposed approach differs from such approaches in that it employs deep learning techniques.

2.4 Identification of Large Classes

Large classes refer to such classes that are doing too much and contain too much code [6]. Such large classes violate

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2019.2936376, IEEE Transactions on Software Engineering

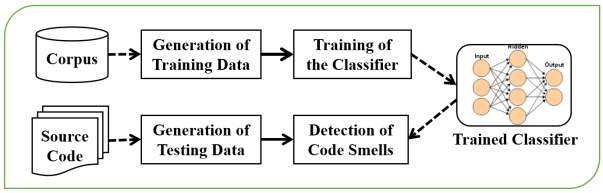


Fig. 1. Overview of the Proposed Approach

single responsibility principle, and thus had better be decomposed into a set of smaller cohesive classes. Refactoring *extract class* could be employed for this purpose.

To identify large classes, a few approaches have been proposed. *iPlasma* proposed by Marinescu et al. [58] according to following formula:

$$(ATFD > few) \wedge (WMC \geq very\_high) \wedge (TCC > one\_third) \quad (7)$$

where ATFD is the *access to foreign data*, WMC is the *weighted method count*, and TCC is the *tight capsule cohesion*. If the expression (formula) holds for a given class, *iPlasma* predicts it as a large class. *DECOR* proposed by Moha et al. [48] identifies large classes by metrics as well:

$$(NMD + NAD) \geq very\_high \quad (8)$$

where NMD is the *number of methods in a class*, and NAD is the *number of attributes in a class*.

2.5 Identification of Misplaced Classes

Packages are frequently employed to organize a group of closely related classes. However, for some reasons, developers may improperly place a class to one package whereas it had better be moved to another package. To solve this problem, we should identify such classes first, and then move them to packages where they should be.

To facilitate the detection of misplaced classes, Palomba et al. [59] propose a textual-based approach. The key insight of their approach is that a well-placed software entity (e.g., class) is often highly similar in vocabulary (i.e., terms extracted from identifiers and comments) with its enclosing entity (e.g., package). Consequently, to investigate whether a given class c should be moved from its enclosing package ep to another package tp, the approach computes the similarity between c and ep (and tp as well). If the similarity between c and ep is smaller than that between c and tp, the given class c is *misplaced* and should be moved to package tp.

3 Approach

In this section, we propose a deep learning based approach to identify code smells. An overview of the proposed approach is presented in Section 3.1, and details are presented in the rest of this section.

3.1 Overview

Fig. 1 presents the overview of the proposed approach. Based on a large corpus of software applications, it generates a huge number of labeled training samples (i.e., smelly and non-smelly software entities). Such training samples are employed to train a neural network based classifier whose output indicates whether the input entity is smelly. Details of the proposed approach are presented in the following sections.

3.2 Generation of Training Data

To train deep neural networks, we generate code smells by smell-introducing refactoring that is defined as follows:

Smell-introducing Refactoring is an unwanted refactoring that reduces software quality.

Applying a refactoring (noted as ar) to well-designed applications would change their well-designed internal structures. As a result, the refactoring leads to bad or sub-optimal design, i.e., code smells. The resulting code smells should be resolved by another software refactoring (noted as ur). Ideally, refactoring ur does nothing but to undo the smell-introducing refactoring ar. An example of smell-introducing refactoring is to move a method from class sc (where the method should be placed) to another class tc. This move method refactoring results in *feature envy* smells. The resulting smells should be resolved by another move method refactoring that moves the method from tc back to sc.

Given a corpus of well-known and high quality open-source applications, we generate positive training data (i.e., smelly software entities) by smell-introducing refactoring as follows:

Step 1: First, from the subject applications, we search for refactoring opportunities (noted as L_{opp}). For example, to create *feature envy* smells, we find all methods that could be moved by *move method* refactoring; to create *long method* smells, we find all method invocations that could be inlined by *inline method* refactoring; to create *large class* smells, we find pairs of classes that could be merged; to create *misplaced class* smells, we find all classes that could be moved by *move class* refactoring.

Step 2: Second, from the resulting refactoring opportunities L_{opp}, we randomly select one (noted as s_{opp}) of them, and apply the corresponding refactoring. Assuming that the subject applications are well-designed, the applied refactoring distorts the original design and thus it is an unwanted refactoring (smell-introducing refactoring). Consequently, it is likely that applying such a smell-introducing refactoring creates code smells. For example, an unwanted *move method* refactoring creates a method associated with *feature envy* smells. This method is then taken as a positive training item. Unwanted *inline method*, *merge class*, and *move class* would result in long methods, large classes, and misplaced classes, respectively.

Step 3: Third, we undo the applied refactoring, remove the selected refactoring opportunity s_{opp} from L_{opp}, and turn to the second step (*Step 2*). The creation of positive training data stops when no more positive items are needed or no more positive item could be generated (i.e., L_{opp} is empty).

On the same corpus of subject applications, we create negative training data as follows:

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2019.2936376, IEEE Transactions on Software Engineering

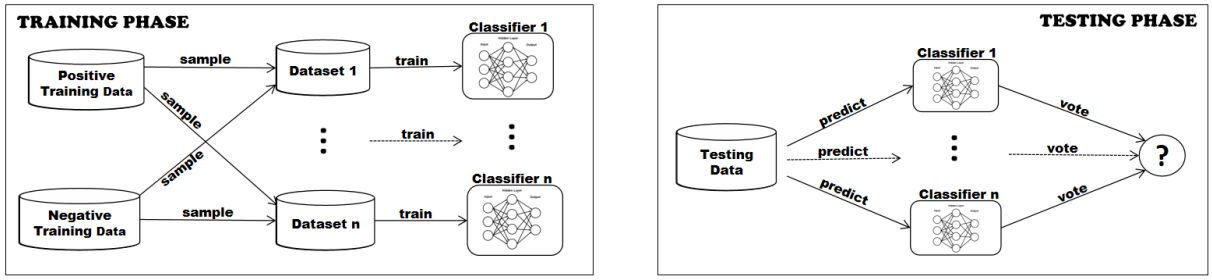


Fig. 2. Common Framework for Deep Learning Based Smell Detection

Step 1: First, from the subject applications, we collect all software entities (noted as L_{se}). The granularity of entities depends on the type of target smells. For example, to create negative examples of *feature envy* and *long method*, we collect all methods. To create negative examples of *large class* and *miss placed class*, however, we should collect classes.

Step 2: Second, we randomly select one entity (noted as e) from L_{se}. Because all elements within the subject applications are well-designed, e should be smell free. Consequently, it could be taken as a negative training item.

Step 3: Third, we remove the selected element e from L_{se}, and turn to the second step (*Step 2*). The creation of negative train data stops when no more negative items are needed or L_{se} is empty.

The generation is based on the assumption that the subject applications are well-designed and all software entities involved in the generation are smell free. However, the assumption may not hold in some cases, and some of the involved software entities may be smelly. As a result, the training data generated based on such entities may be noisy: some of them are labeled incorrectly. The impact of such noisy data could be reduced in two ways. First, we only select high quality subject applications for data generation. Within such high quality applications, it is likely that most of the involved software entities are smell free. As a result, most of the training items could be labeled correctly. Second, advanced neural networks work well even if the training data contain a few mislabeled items [60].

3.3 Deep Learning Based Smell Detection

With a large amount of labeled training data, we may train complex neural network based classifiers to identify different categories of code smells. However, we need different features to detect different categories of code smells, and thus it is challenging to design (and train) a generic classifier to detect all categories of code smells. To this end, we present different classifiers for different code smells in Sections 4-7. In this section, we only introduce the common framework for such classifiers.

The common framework is presented in Fig. 2. The left part of the figure presents the training phase whereas the right part presents the prediction (testing) phase. The positive and negative training data in Fig. 2 refer to those generated in Section 3.2. From such data sets, we randomly sample a subset of them to form a training dataset (noted as $dataset_1$) and train a neural network based classifier with the dataset. The resulting classifier is noted as $classifier_1$. We repeat the process (sampling and training) for n times, which results in n classifiers.

For a given software entity to be detected, we feed it into each of the resulting classifiers (from $classifier_1$ to $classifier_n$), and each of them generates a binary classification (i.e., smelly or non-smelly). Finally, these classifiers make the final decision by voting. We train multiple classifiers and make final decisions by voting to improve the stability and accuracy of the proposed smell detection algorithms. The approach (i.e., training multiple classifiers and making final decisions by voting) is also known as bootstrap aggregating [25], [26]. It is a machine learning ensemble meta-algorithm that has been proved helpful in improving the stability and accuracy of machine learning algorithms [25]. Evaluation results in Section 8 also confirm that it does in improve the performance of the proposed approach.

4 Deep Learning Based Feature Envy Detection

As an initial application of the generic approach proposed in Section 3, we apply it to the detection of *feature envy* in this section. Feature envy refers to misplaced methods that should be moved by *move method* refactorings to classes where they should be [6]. It is one of the most common code smells [24].

4.1 Feature Selection

To decide whether a given method m should be moved from its enclosing class ec to another class tc, we exploit both structural information (code metrics) and textual information. Concerning the structural information, we reuse the distance metrics (as presented in Formula 3 and Formula 4) proposed by Tsantalis and Chatzigeorgiou [39]. We reuse such metrics because of the following reasons:

- First, they have been proved effective in feature envy detection [39].
- Second, the open-source implementation of *JDeodorant* makes it easy to extract such metrics.

Besides the metrics, we also exploit textual information, including the name of the method to be investigated, the name of its enclosing class, and the name of its potential target class. The essence of feature envy is that some methods are misplaced. Ideally, a method should be declared

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2019.2936376, IEEE Transactions on Software Engineering

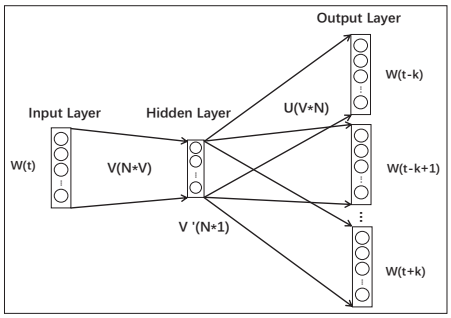


Fig. 3. Model of Word2Vector

within the class whose role should have the behavior of the method. We may identify whether a given method should be declared within a given class by investigating the semantical relationship between their identifiers because meaningful identifiers can reveal the roles /behaviors of the related entities [61].

As a conclusion, the input of the approach is a quintuple:

$$input = < name(m), name(ec), name(tc), dist(m, ec), dist(m, tc) > \quad (9)$$

where $name(e)$ is the identifier (method name or class name) of software entity e. m is the method under investigation, ec is the enclosing class of m, and tc is the potential target class. $dist(m, c)$ is the distance between method m and class c computed according to Formula 3 or Formula 4.

However, it is challenging to recover automatically the semantical relationship embedded in method names and class names, i.e., $name(m)$, $name(ec)$, and $name(tc)$. Lexical similarity alone is often insufficient in measuring the semantical relationship between software entities. For example, lexically dissimilar software entities (e.g., *CollectCandidates* and *RecommenderSystem*) could be closely related in semantics. Consequently, to fully exploit the semantics embedded in natural languages, we should employ some advanced technologies, e.g., deep learning, to extract more useful features from such textual input. Besides that, it is also challenging to quantitatively relate textual features to numerical feature (code metrics) with handcrafted heuristics. The proposed approach handles these challenging issues in following sections.

4.2 Representation of Identifiers

To feed the identifiers described in nature languages into neural networks, we convert words in identifiers into fixed-length numerical vectors. The conversion is accomplished by the well-known *word2vector* (continuous skip gram) proposed by Mikolov et al. [62], [63]. *Word2vector* has been proved efficient for learning high-quality distributed vector representations that capture precise syntactic and semantic word relationships [62]. *Word2vector* is essentially a neural network that predicts nearby words, i.e., words before and after it (as shown in Fig 3). Once the network is trained, we can exploit the hidden layer, a byproduct of the training, to convert words into numerical vectors.

For a given identifier (method name or class name), we partition it into a sequence of words according to capital letters and underscores, and convert each word into a fixed-length numerical vector:

$$name(e) = < w_1, w_2, \ldots, w_k > \quad (10)$$
$$= < V(w_1), V(w_2), \ldots, V(w_k) > \quad (11)$$

$name(e)$ is the identifer of software entity e, and $< w_1, w_2 \ldots, w_k >$ is a sequence of words. $V(w_i)$ converts word w_i into a fixed-length (200 dimensions) numerical vector with *word2vector*.

To facilitate the design of neural networks, we limit the length of word sequence for each identifier to five. Our analysis results on open-source applications (as introduced in Table 1) suggest that 98.5%=(184,613/187,377) of the involved identifiers contain no more than five words. If an identifier contains more than five words, we extract the first five words only. In contrast, if it contains less than five words, we append special characters (whose vectors are composed of zeros only) to the sequence.

4.3 Training Data for Feature Envy Detection

To apply deep learning techniques to code smell detection, we generate training data as follows. First, we download well-known and high quality open-source applications. Second, for each method m (excluding testing methods) from such applications, we generate a labeled training sample as follows. (1) We test whether m could be moved to other classes with move method refactoring. The test is accomplished with the APIs provided by Eclipse JDT: It considers parameters and fields accessed in m as potential references to target classes, and thus takes such classes as potential target classes (noted as ptc) for m. If ptc is empty, method m could not be moved. Otherwise, it could be moved to any of the classes within ptc. (2) Suppose that method m could be moved to a set of classes noted as $ptc = \{tc_1, tc_2, \ldots, tc_k\}$. If ptc is empty, i.e., the method could not be moved, we discard it and turn to the next method. Otherwise, we turn to the next step to generate a labeled training item. (3) We randomly (fifty-fifty chance) decide to generate a positive training item (with feature envy) or a negative item (without feature envy). (4) We generate a negative item as follows. First, we randomly select a potential target class tc_i from ptc. Second, we compute the distance $dist(m, ec)$ and $dist(m, tc_i)$, where ec is the enclosing class of m. Third, we create a negative item ($ngItem$) and add it to the training data set:

$$ngItem = < input, output > \quad (12)$$
$$input = < name(m), name(ec), name(tc_i), dist(m, ec), dist(m, tc_i) > \quad (13)$$
$$output = 0 \quad (14)$$

(5) We generate a positive item as follows. First, we randomly select a potential target class tc_i from ptc. Second, we move m from its enclosing class ec to tc_i by Eclipse APIs. Third, we create a positive item whose input is $< name(m), name(tc_i), name(ec), dist(m, tc_i), dist(m, ec) >$ and output is 1. Notably the distances are computed after the method is moved.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2019.2936376, IEEE Transactions on Software Engineering

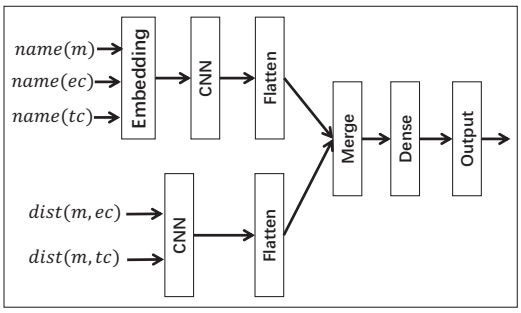


Fig. 4. Classifier for Feature Envy Detection

4.4 Classifier for Feature Envy Detection

The structure of the deep neural network based classifier for feature envy detection is presented in Fig. 4 and source code is available online [64]. Its input is divided into two parts: textual input and numerical input. The textual input is a word sequence by concatenating the name of the method, the name of its enclosing class, and the name of the potential target class. It is fed into an embedding layer that converts text description into numerical vectors as introduced in Section 4.2. Such numerical vectors are in turn fed into a Convolutional Neural Network (CNN). In total we have two CNN layers whose setting is as follows: $filters = 128$, $kernel \quad size = 1$ and $activation = tanh$. We exploit CNN because of the following reasons. First, significant advances in CNN have been achieved recently, which makes CNN effective in increasing the capacity and flexibility of machine learning [65]. Powerful CNN layers may learn the deep semantical relationship among the identifiers, and thus may reveal where the method should be placed. Second, CNN is well-suited for parallel computation on modern powerful GPU, and thus can significantly reduce training time [66]. The output of CNN is forwarded to a *flatten* layer [67] which turns its input into a one-dimensional vector.

The numerical input, i.e., $dist(m, ec)$ and $dist(m, tc)$, is fed directly into another CNN whose output is forwarded to a *flatten* layer. This CNN shares the same setting with the previous CNN introduced in the preceding paragraph. Notably, we have experimentally tried to replace this CNN with other types of neural network layers (e.g., dense layer), but we fail to improve the performance with the replacement. The textual input and the numerical input are finally merged by the *merge* layer [68] which simply concatenates a list of inputs (i.e., the aforementioned textual and numerical inputs). The following *dense* layer (128 neurons) and *output* layer (2 neurons) map the textual input and numerical input into a single output (prediction) that indicates whether m should be moved to the target class tc. The model employs $binary\_crossentropy$ as the loss function.

4.5 Feature Envy Detection

4.5.1 Binary Classification

For a given method m to be investigated, we predict whether it is smelly or non-smelly as follows. First, we collect all of its potential target classes (noted as $ptc = \{tc_1, tc_2, \ldots, tc_k\}$) with Eclipse JDT. If ptc is empty, i.e., the method could not be moved, m is not smelly. Otherwise, we generate a testing item (as shown in Formula 9) for each of the potential classes (noted as tc_i): $input_i =< name(m), name(ec), name(tc_i), dist(m, ec), dist(m, tc_i) >$ where ec is the enclosing class of method m. We feed such items into the trained deep neural network. If all of such items are predicted as negative (non-smelly), we say that the given method m is not associated with feature envy. Otherwise, we say that it is smelly (associated with feature envy).

4.5.2 Recommendation of Refactoring Solutions

For methods that are predicted as smelly (with feature envy), we should suggest where such methods should be moved via move method refactorings. If only one (noted as $input_j$) of the testing items generated for m is predicted as positive, we suggest to move m to the target class (tc_j) that is associated with the positive testing item $input_j$.

If more than one testing items are predicted as positive, we select the one (noted as $input_i$) with the greatest output, and suggest to move method m to class tc_i that is associated with $input_i$. Although the neural network proposed in Section 4.4 is trained as a binary classifier (aiming to minimize the likelihood of misclassification instead of the mean squared error), the output of the neural network is a decimal varying from zero to one. The neural network interprets the prediction as positive if and only if the output is greater than 0.5 [69]. For the given input $input =< name(m), name(ec), name(tc_i), dist(m, ec), dist(m, tc_i) >$, the greater the output is, the more likely that the m should be moved to the target class tc_i.

5 Deep Learning Based Long Method Detection

As another application of the generic approach proposed in Section 3, we apply it to the detection of *long method*. Long methods refers to those methods that are complex and lack of cohesion [6]. Such methods are difficult to read, maintain, or reuse [52]. Consequently, such methods had better be decomposed into smaller and more cohesive ones [55].

5.1 Metrics for Long Method Detection

To decide whether a given method m is associated with *long method*, Sofia et al. [47] transform the following metrics to method-level: Lines of Code (LOC), Lack of Cohesion of Methods (LCOM1, LCOM2, and LCOM4), Cohesion (COH), and Class Cohesion (CC). Their evaluation results suggest that such metrics are promising in detecting long methods [47]. Consequently, we reuse such metrics for long method detection. Besides those, we also employ the Number of Accessed Variables (NOAV), McCabe's Cyclomatic Number (MCN), and Coupling Dispersion (CD) [70]. NOAV and MCN measure the complexity of the method whereas CD measures how much the coupling of the method involves external classes. As a result, for a given method (noted as m) to be detected, we generate the follow item as the input to the neural network based classifier:

$$\begin{aligned} feature(m) \quad = \quad & < LOC(m), LCOM1(m), LCOM2(m), \\ & LCOM4(m), COH(m), CC(m), \\ & NOAV(m), CD(m), MCN(m) > \end{aligned} \quad (15)$$

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2019.2936376, IEEE Transactions on Software Engineering

5.2 Training Data for Long Method Detection

we generate training data for long method detection as follows. First, we download well-known and high quality open-source applications. Second, for each method m from such applications, we try to generate a labeled training sample as follows. (1) We validate whether *Inline Method* refactoring could be conducted on any of the method invocations within m. Method invocations may be unsuitable for *inline* because of various reasons, e.g., recursive methods, overriding method, and constructors. If there is no *inline method* refactoring opportunity within m, we turn to the next method. (2) If more than one *inline method* refactoring opportunities are found within m, we select one of them according to the following rules:

R_1:Methods implemented in other classes are in preference to those implemented within the enclosing class of m;

R_2: If multiple methods have the same priority according to R_1, the longest one is preferred;

R_3: If multiple methods have the same priority according to R_1 and R_2, one of them is randomly selected.

(3) We randomly (fifty-fifty chance) decide to generate a positive training item or a negative item. (4) We generate a negative item by taking the intact method m. The following negative item ($ngItem$) is added to the training data set:

$$ngItem = < feature, flag > \quad (16)$$

$$feature(m) = < LOC(m), LCOM1(m), LCOM2(m), LCOM4(m), COH(m), CC(m), NOAV(m), CD(m), MCN(m) > \quad (17)$$

$$flag = 0 \quad (18)$$

(5) To generate a positive item, we apply the *inline method* refactoring opportunity selected in preceding steps. After that we create a positive item by taking the refactored method m. Notably, all metrics (LOC, LCOM1, LCOM2, LCOM4, COH, CC, NOAV, MCN, and CD) of the method are computed after the *inline method* refactoring.

5.3 Classifier for Long Method Detection

The structure of the deep neural network based classifier for long method detection is presented in Fig. 5 and source code is available online [64]. It is composed of five dense layers besides an input layer and an output layers. The resulting four features extracted by the hidden layers are fed into the output layer that maps the features into a single output (prediction) to suggest whether m is associated with long method smells.

Notably, we have experimentally tried to replace the dense layers with other types of neural network layers. Recently, more advanced structures, like CNN and RNN, have been proposed and have been proven superior to dense layers in many tasks, e.g., image processing and natural language processing. Consequently, replacing the dense layers (hidden layers) with such advanced structures may further improve the performance of the proposed approach. However, in our initial experiments, we fail to improve the performance with the replacement. For example, replacing the dense layers with CNN results in slight reduction in F_1 score of the proposed approach. One of the possible reasons

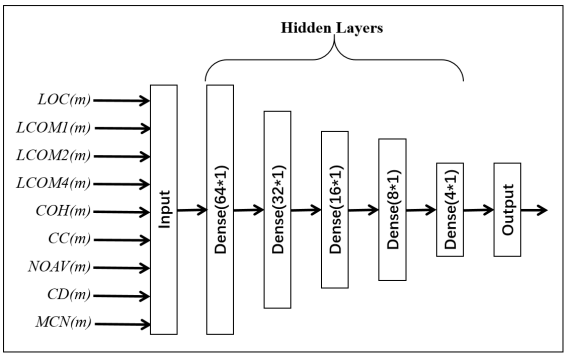


Fig. 5. Classifier for Long Method Detection

is that the input of the classifier (i.e., a set of code metrics) do not have obvious receptive fields (the key concept for CNN) or temporal relationship (the key concept for RNN). Another possible reason is that the number of input is small, and thus the number of parameters within the neural network is relatively small even if we employ the dense layers only. Consequently, the resulting network could be trained efficiently and effectively, which makes the dense layer based classifier promising.

For a given method m to be investigated, we predict whether it is smelly or non-smelly as follows. First, we compute its metrics, including LOC, LCOM1, LCOM2, LCOM4, COH, CC, NOAV, CD, and MCN. Second, we generate a testing item (as shown in Formula 15): $it =< LOC(m), LCOM1(m), LCOM2(m), LCOM4(m), COH(m), CC(m), NOAV(m), CD(m), MCN(m) >$. We feed such item into the trained classifier in Fig. 5. If the item is predicted as negative (non-smelly), the given method m is not a long method. Otherwise, it is and it should be decomposed into smaller ones.

6 Deep Learning Based Large Class Detection

As the third application of the generic approach proposed in Section 3, we apply it to the detection of *large class* in this section. Large classes are doing too much and contain too much code [6]. Consequently, such large classes should be decomposed into a set of smaller cohesive classes by *extract class* refactoring [6].

6.1 Feature Selection for Large Class Detection

The selected features are composed of two parts: textual information and code metrics. Textual information is the identifiers (field names and method names) declared within the class to be tested. Code metrics include 12 class-level common code metrics [71], [70], [72], i.e., ATFD (Access To Foreign Data), DCC (Direct Class Coupling), DIT (Depth of Inheritance Tree), TCC (Tight Class Cohesion), LCOM (Lack of Cohesion in Methods), CAM (Cohesion Among Methods of Class), WMC (Weighted Method Count), LOC (Lines of Code), NOPA (Number of Public Attributes), NOAM (Number of Accessor Methods), NOA (Number of Attributes), and NOM (Number of Methods). Consequently, the selected

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2019.2936376, IEEE Transactions on Software Engineering

features for class c could be presented as $slFetures(c)$ as follows:

$$slFetures(c) =< TexInfo(c), cdMetrics(c) > \quad (19)$$

$$TexInfo(c) =< Attributes(c), Methods(c) > \quad (20)$$

$$Attributes(c) =< AttributeName_1(c), AttributeName_2(c), \ldots, AttributeName_n(c) > \quad (21)$$

$$Methods(c) =< MethodName_1(c), MethodName_2(c), \ldots, MethodName_m(c) > \quad (22)$$

$$cdMetrics(c) =< ATFD(c), DCC(c), DIT(c), TCC(c), LCOM(c), CAM(c), WMC(c), LOC(c), NOPA(c), NOA(c), NOM(c) > \quad (23)$$

where $TexInfo(c)$ is the textual information, $cdMetrics(c)$ is the code metrics of class c, $AttributeName_i(c)$ is the name of the ith attributed defined by c, and $MethodName_i(c)$ is the name of the ith method defined by c. The names of attributes and methods are turned into numerical vectors by *word2vector* as introduced in Section 4.2

6.2 Training Data for Large Class Detection

We generate training data for large class detection as follows. First, we download well-known and high quality open-source applications. Second, for each pair of classes from the same application, we try to merge them as a single large class. If the mergence succeeds, the resulting class is taken as positive example (i.e., large class that should be decomposed). If the mergence fails for various reasons (e.g., multiple inheritance), we undo the mergence. Notably, for each class in the subject applications, they should be involved in the resulting data for no more than once. Consequently, if the mergence of two classes succeeds, they would not be employed for training data generation any more. We create as many positive examples as we can on such applications. Finally, we generate negative examples by randomly selecting classes from such applications, assuming that such classes have been well-designed and they do not deserve further decomposition. The number of negative examples is equal to that of positive examples we create in the preceding step.

6.3 Classifier for Large Class Detection

Similar to the classifiers for *feature envy* and *long method*, the classifier for *large class* is also based on deep neural network. The structure is presented in Fig. 6 and the source code of the classifier is available online [64]. Its input is composed of two parts: textual input and numerical input (code metrics). The textual input is a word sequence by concatenating the name of attributes and methods declared within the class under test (noted as c). The textual input is converted into numerical vectors as introduced in Section 4.2 by the embedding layer. The resulting vectors are handled by a Long Short-Term Memory (LSTM) layer. In contrast, the code metrics are fed into a dense layer directly. Its output is merged with the output of LSTM before they are fed into another dense layer. The output of the dense layer indicates whether the class under test should be decomposed or not.

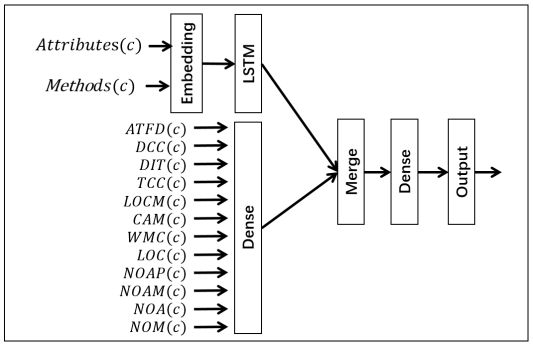


Fig. 6. Classifier for Large Class Detection

7 Deep Learning Based Detection of Misplaced Classes

As the fourth application of the generic approach proposed in Section 3, we apply it to the detection of *misplaced class* in this section. Misplaced classes refer to such classes that are placed in wrong positions (packages). Such classes should be moved to the packages where they should be.

7.1 Features for Misplaced Class Detection

To decide whether a given class c should be moved from its enclosing package ep to another package tp, we leverage two categories of features: code metrics and textual features. Employed code metrics include CBO (Coupling Between Objects) and MPC (Message Passing Coupling) [70]. $CBO(c, ep)$ is the number of classes from package ep that class c refers to. $MPC(c_1, c_2)$ is the number of method calls defined in methods of class c_1 to methods in class c_2. The selected features for misplaced class detection could be presented as $fs(c, ep, tp)$ as follows:

$$fs(c, ep, tp) =< tfs(c, ep, tp), mfs(c, ep, tp) > \quad (24)$$

$$tfs(c, ep, tp) =< Name(c), Name(ep), Name(tp) > \quad (25)$$

$$mfs(c, ep, tp) =< CBO(c, ep), maxMPC(c, ep), avgMPC(c, ep), sumMPC(c, ep), CBO(c, tp), maxMPC(c, tp), avgMPC(c, tp), sumMPC(c, tp) > \quad (26)$$

$$maxMPC(c, ep) = \max_{c_i \in ep} MPC(c, c_i) \quad (27)$$

$$sumMPC(c, ep) = \sum_{c_i \in ep} MPC(c, c_i) \quad (28)$$

$$avgMPC(c, ep) = \frac{1}{|ep|} \sum_{c_i \in ep} MPC(c, c_i) \quad (29)$$

where $Name(c)$, $Name(ep)$, and $Name(tp)$ are names of class c, package ep, and package tp, respectively. $|ep|$ is number of classes defined within package ep. Notably, the names of classes and packages are turned into numerical vectors by *word2vector* as introduced in Section 4.2

7.2 Training Data for Misplaced Class Detection

We generate training data for misplaced class detection as follows. First, we download high quality open-source applications from Github. Second, for each class c_i in the subject applications, we validate whether it could be moved to other classes. If yes, we decide to generate positive or

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2019.2936376, IEEE Transactions on Software Engineering

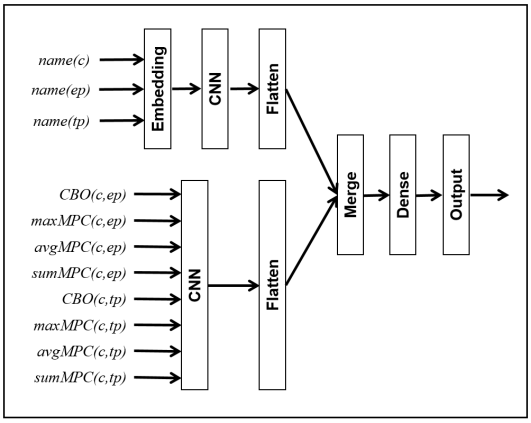


Fig. 7. Classifier for Misplaced Class Detection

negative item (fifty-fifty chance) based on this class. Third, to create a positive item, we randomly pick up one of the target packages to which class c_i could be moved. After the movement, class c_i becomes a misplaced class. Fourth, to create a negative item, we just take the untouched c_i as the negative item assuming that it is well-placed.

7.3 Classifier for Misplaced Class Detection

The structure of the deep neural network based classifier for misplaced class is presented in Fig. 7 and source code is available online [64]. The structure of the classifier is highly similar to that for feature envy detection as presented in Fig. 4. The rationale for the similarity is that both feature envy and misplaced classes are caused by misplaced software entities (methods and classes, respectively). Consequently, they could be identified by similar classifiers.

The textual input, i.e., a word sequence by concatenating the name of the class, the name of its enclosing package, and the name of the potential target package, is converted into numerical vectors by the embedding layer. Details of the conversion are presented in Section 4.2. The resulting vectors are handled by a CNN layer. After that a *flatten* layer [67] turns the output of the CNN into a one-dimensional vector.

We employ another CNN layer (the bottom left part of Fig 7) to handle the numerical input, i.e., $CBO(c, ep)$, $maxMPC(c, ep)$, $avgMPC(c, ep)$, $sumMPC(c, ep)$, $CBO(c, tp)$, $maxMPC(c, tp)$, $avgMPC(c, tp)$, and $sumMPC(c, tp)$). Its output is also forwarded to a *flatten* layer. The output of both *flatten* layers is merged by a *merge* layer [68]. The last two layers (a *dense* layer and a *output* layer) generate the single output (i.e., final prediction).

7.4 Misplaced Class Detection

Based on the classifier presented in Section 7.3, we can identify misplaced classes (i.e., smell detection) and recommend where such classes should be moved (i.e., refactoring solutions). For a given class c that is suspiciously misplaced, we decide whether it is misplaced as follows:

- First, we collect all packages to which c could be moved, noted as $ptp(c)=\{tp_1(c), tp_2(c), \ldots, tp_k(c)\}$. If $ptp(c)$ is empty, c is not smelly because its current enclosing package is the only place where it could be settled. In this case, the detection on c terminates. Otherwise, it turns into the following steps.
- Second, we generate a testing item $fs_j(c, ep, tp_j(c))$ (as defined in Formula 24) for each of the potential packages (noted as $tp_j(c) \in ptp(c)$) where ep is the enclosing package of class c.
- For each of the testing items, we test it with the classifier in Fig. 7. Class c is a misplaced class if and only if any of the testing items is predicted as positive.

Although the neural network is trained as a binary classifier, its actual output is a decimal varying from zero to one. For testing item $fs_j(c, ep, tp_j(c))$, the greater the output of the network (noted as $o(fs_j(c, ep, tp_j(c)))$) is, the more likely it is that class c should be moved to package $tp_j(c)$. Consequently, for an identified misplaced class c, we recommend to move it to package $tp_i(c) \in ptp(c)$ (called *target package*) if:

$$o(fs_j(c, ep, tp_i(c))) = \max_{tp_j \in ptp(c)} o(fs_j(c, ep, tp_j(c))) \quad (30)$$

If there exist multiple target packages that have equal network output, we randomly select one of them as the recommended target package.

8 Evaluation

In this section, we evaluate the proposed approach on ten open-source applications with injected code smells.

8.1 Research Questions

The evaluation investigates the following research questions:

- **RQ1**: Does the proposed approach outperform the state-of-the-art approaches in identifying feature envy?
- **RQ2**: Is the proposed approach accurate in recommending destinations (target classes) for methods associated with feature envy?
- **RQ3**: Does the proposed approach outperform the state-of-the-art approaches in identifying long methods?
- **RQ4**: Does the proposed approach outperform the state-of-the-art approaches in identifying large classes?
- **RQ5**: Does the proposed approach outperform the state-of-the-art approaches in identifying misplaced classes?
- **RQ6**: Is the proposed approach accurate in recommending target packages for misplaced classes?
- **RQ7**: How efficient is the proposed approach? How long does it take to train the neural network based classifier, and how long does it take to make predictions?
- **RQ8**: How does bootstrap aggregating influence the performance of the proposed approach?

Research question *RQ1* investigates the performance (e.g., precision and recall) of the proposed approach in identifying feature envy smells compared against the state-of-the-art approaches. To answer this question, we compare the proposed approach against a special version of *JDeodorant* [39]. Notably, *JDeodorant* is a refactoring tool who recommends refactoring opportunities instead of simply reporting

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2019.2936376, IEEE Transactions on Software Engineering

code smells. To this end, for each identified code smell, it would try to figure out refactoring solutions (noted as pre-condition checking). If it fails to figure out any solutions, it would not report the related code smells. To make fair comparison, during the evaluation we disable such pre-condition checking of *JDeodorant*. We select *JDeodorant* for comparison because of the following reasons. First, it represents the state-of-the-art, and has been widely employed as a benchmark [41], [44], [36]. Second, it is publicly available. Although some related approaches have been reported recently, their implementations are not available, which makes it difficult to compare the proposed approach against such approaches.

Research question *RQ2* concerns how accurate the proposed approach is in recommending where the feature envy methods should be moved. The ultimate goal of code smell detection is not to find out code smells, but to remove such smells by software refactorings and thus to improve software quality. Consequently, it is critical for the proposed approach to suggest correctly to which classes the misplaced methods should be moved.

Research question *RQ3* investigates the performance of the proposed approach in identifying long methods compared against the state-of-the-art approaches. To answer this question, we compare the proposed approach against *DECOR* proposed by Moha et al. [48]. Research question *RQ4* investigates the performance of the proposed approach in identifying large classes compared against the state-of-the-art approach (*DECOR* [48]).

Research question *RQ5* investigates how accurate the proposed approach is in identifying misplaced classes. Because neither *DECOR* nor *JDeodorant* are designed to identify misplaced classes, we compare the proposed approach against another code smell detection tool *TACO* proposed by Palomba et al. [59] in the identification of misplaced classes. It was released recently, and has been proved highly accurate [59]. Research question *RQ6* investigate how accurate the proposed approach is in recommending refactoring solutions (i.e., target packages) for misplaced classes.

Research question *RQ7* concerns the time complexity of the proposed approach. Deep learning based approaches often take a long time to train deep neural networks. However, they usually response instantly for a given input once the neural networks are trained in advance. Answering this question would reveal quantitatively the training and predicting time of the proposed approach.

Research question *RQ8* concerns the influence of bootstrap aggregating. It is widely employed to generate strong classifiers based on weak ones [25], [26]. Answering this question would reveal quantitatively how it improves the performance of the proposed approach.

8.2 Subject Applications

We evaluate the proposed approach on ten open-source applications as introduced in Table 1. Although the proposed approach is generic and should be able to work for different object-oriented programming languages, its prototype implementation is confined to Java only. Consequently, we select Java applications only. The columns (from left to right) present the application name, domain, version, num of classes (NOC), number of methods (NOM), and lines of source code (LOC), respectively.

TABLE 1
Subject Applications

Applications	Domain	Version	NOC	NOM	LOC
JUnit	Unit Testing	4.10	123	866	11,734
PMD	Static Code Analysis	5.2.0	250	2,097	32,783
JExcelAPI	Excel API	2.6.12	424	3,118	90,555
Areca	Document Backup	7.4.7	473	5,055	88,126
Freeplane	Knowledge Management	1.3.12	787	6,938	124,937
jEdit	Text Editor	4.5.0	513	5,964	185,571
Weka	Machine Learning	3.9.0	1348	20,182	444,493
AbdExtractor	Android	20140630	1695	12,608	304,458
Art of Illusion	3D modelling	3.0	492	6,188	152,207
Grinder	Testing	3.6	502	3,037	101,293

JUnit [73] is a widely used testing framework. During the last 16 years, it has released more than 40 versions. *PMD* [74] is an extensible cross-language static code analyzer that is widely used to find common programming flaws. During the last 14 years, it has evolved extensively and released more than 100 versions. *JExcelAPI* [75] is an open-source Java API that facilitates developers to read, write or modify *Excel* spreadsheets dynamically. The evolution lasted more than 7 years, and 75 versions have been released. *Areca* [76] is an open-source application for file backup, supporting incremental backup on local drives or FTP servers. The evolution lasted more than 8 years, and 90 versions have been released. *Freeplane* [77] is a free mind mapping and knowledge management software. During the last 18 years, it has released more than 100 versions. *jEdit* [78] is a free text editor. During the last 18 years, it has released 143 versions. *Weka* [79] is a set of well-known machine learning algorithms developed by the machine learning group at the University of Waikato. The evolution lasted more than 18 years, and more than 90 versions have been released. *Android Backup Extractor* [80] (also known as *AbdExtractor*) is a utility to extract and repack Android backups. During the last 7 years, 22 versions have been released. *Art of Illusion* [81] is an open source 3D modelling and rendering studio. Its evolution lasted more than 20 years, and developers have released more than 40 versions. *Grinder* [82] is an open source Java load testing framework. Since *Grinder* 2 was released in 2013, it has gone through more than 70 versions.

These subject applications are selected because of the following reasons. First, all of them are open-source applications whose source code is publicly available. Selecting such open-source applications facilitates other researchers to repeat the evaluation. Second, all of them are well-known,

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2019.2936376, IEEE Transactions on Software Engineering

popular and of high quality. As introduced in Section 4.3, the generation of training data is based on the assumption that software entities in subject applications are well designed. All of the subject applications have involved successfully for a long time (more than 5 years), which usually depends on high quality of the maintained projects. We also manually checked candidate projects to make sure that the selected projects are of high quality. Notably, it is not straightforward to assess the design quality of a project, and thus the manual checking is rather informal: The authors read through sample files just like code reviewers, and informally assess the quality of the projects. Besides that, we also employ SonarQube Quality Gate metrics[1] to assess the quality. The checking results suggest that all of the subject applications have passed the quality checking, and received "A" rating concerning code smells. Third, these applications were developed by different developers, which may reduce the bias introduced by specific developers.

8.3 Process

8.3.1 Evaluation on Feature Envy Detection

We carry out leave-one-out validation on the ten subject applications presented in Table 1. When a single application is used as the testing subject (noted as *testingApp*), the others are used as training subjects (noted as *trainingApps*). Each of the subject applications is used as the testing subject for once. We employ this validation method because we expect the proposed approach to be trained in advance (offline) with data generated from existing applications, and the training should not exploit any data from the testing application. Otherwise, we have to re-train the model before prediction, which may prevent the proposed approach from instant prediction.

The evaluation on feature envy detection follows the following process:

1) First, we generate a training data set $trainingData$ based on *trainingApps* as specified in Section 4.3.
2) Second, we train the proposed approach with *trainingData* .
3) Third, from the testing subject *testingApp*, we identify all methods (noted as ms) that could be moved between classes via *JDT' move method* API.
4) Fourth, from ms we randomly sample 23% of the methods, and note them as ms_1. The rest is noted as ms_2, and thus we have $ms = ms_1 \bigcup ms_2$.
5) Fifth, for each method m in ms_1, we randomly move it to one of its potential target classes (i.e., injecting smells into *testingApp*).
6) Sixth, after the movement, we apply the proposed approach and *JDeodorant* to the testing subject independently. A predicted positive is a true positive if and only if the reported method m has been moved before (i.e., $m \in ms_1$).

Methods may be unmovable because of differen reasons [83]. For example, moving overridden/overriding methods often leads to syntax errors. Another example is that moving method m to another class that is not related to its enclosing class in any way may break the links (e.g., method invocation) between m and members of its enclosing class. Such unmovable methods could not serve as move method opportunities. Smell detection tools can also exclude such methods by static analysis or simply by existing JDT APIs. Consequently, during the evaluation we do not generate testing items based on such methods. Instead, we generate testing items based on movable methods only, although analysis on the subject applications introduced in Table 1 suggests that only 10%(=4,430/44,220) of the methods could be moved.

We also notice that experimental study conducted by Palomba et al. [84] suggest that around 2.3% of methods are associated with feature envy and should be resolved by move method refactorings. Consequently, from the movable methods, we randomly select 23% of them to construct positive testing items. As a result, the smelly methods account for around 2.3%=$10\% \times 23\%$ of all methods (including unmovable ones), which is consistent with the finding reported by Palomba et al. [84].

8.3.2 Evaluation on Long Method Detection

Notably, the evaluation of the proposed approach on feature envy detection and long method detection is conducted independently. Similar to the evaluation on feature envy detection, we also conduct leave-one-out validation on the subject applications presented in Table 1 to evaluate the proposed approach on long method detection. Each fold of the evaluation follows the following process:

1) First, we generate a training data set $trainingData$ based on *trainingApps* as specified in Section 5.2.
2) Second, we train the proposed approach with *trainingData* .
3) Third, we inject *long method* smells into the testing subject *testingApp*. Although Palomba et al. [84] mention the ratio of methods associated with feature envy, they do not explicitly mention the ratio of other code smells, e.g., long methods, large classes, and misplaced classes. To figure out the ratios in real projects, we ask three developers to manually check the first three subject applications (i.e., *XMD*, *JSmooth*, and *Neuroph*) involved in the case study in Section 9. We call the manual checking *Pilot Checking*. The pilot checking reveals that around 12% of the methods are long methods. Consequently, the injection guarantees that smelly methods account for around 12% of the methods in the resulting subject applications.
4) Finally, after the injection, we apply the proposed approach and *DECOR* to the testing subject independently. A predicted positive is a true positive if and only if the reported method has been injected code smells by *inline method* refactoring.

8.3.3 Evaluation on Large Class Detection

The evaluation process on large class detection is identical to that on long method detection except for the following aspects:

1) First, the training data set is constructed in the way specified in Section 6.2.
2) Second, the smell injection guarantees that injected smelly classes (i.e., large classes) account for around 6%

1. https://www.sonarqube.org/downloads/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2019.2936376, IEEE Transactions on Software Engineering

TABLE 2
Evaluation Results on Feature Envy Detection

Applications	Proposed Approach					JDeodorant				
	$precision$	$recall$	F_1	MCC	AUC	$precision$	$recall$	F_1	MCC	AUC
Areca	50.00%	87.96%	63.76%	53.89%	91.93%	24.24%	6.78%	10.60%	2.27%	40.81%
Freeplane	36.24%	94.14%	52.33%	36.69%	83.10%	34.91%	12.63%	18.55%	7.41%	48.67%
jEdit	38.18%	91.30%	53.85%	39.42%	84.29%	37.50%	9.74%	15.46%	8.45%	46.96%
JUnit	50.00%	82.22%	62.18%	46.02%	85.72%	61.90%	24.53%	35.14%	27.55%	45.36%
PMD	37.37%	86.05%	52.11%	36.75%	84.90%	50.00%	20.00%	28.57%	24.10%	52.85%
Weka	38.24%	87.00%	53.13%	40.22%	78.74%	72.36%	31.93%	44.31%	40.17%	52.89%
AbdExtractor	29.70%	76.67%	42.82%	21.66%	74.82%	30.77%	11.31%	16.54%	6.48%	51.36%
Grinder	31.20%	88.64%	46.15%	35.51%	85.21%	40.00%	6.45%	11.11%	10.77%	51.70%
Art of Illusion	36.68%	96.97%	53.22%	51.54%	93.63%	18.75%	6.43%	9.57%	4.14%	37.06%
JExcelAPI	34.04%	88.89%	49.23%	39.54%	89.98%	50.00%	7.69%	13.33%	13.45%	61.24%
Average	**36.79%**	**88.11%**	**51.91%**	**39.63%**	**84.90%**	**46.87%**	**16.60%**	**24.51%**	**18.37%**	**50.18%**

of the classes in the resulting subject applications, the same ratio as we find in the subject applications in the *Pilot Checking* (as introduced in Section 8.3.2).

8.3.4 Evaluation on Misplaced Class Detection

The leave-one-out validation of misplaced class detection is similar to that of feature envy detection. It is also composed of two steps: smell detection and refactoring recommendation (i.e., where the misplaced class should be moved). In the first step, the training data set is constructed in the way specified in Section 7.2. Notably, the smell injection guarantees that injected smelly classes (i.e., misplaced classes) account for around 15% of the classes in the resulting subject applications, the same ratio as we find in the subject applications in the *Pilot Checking* (as introduced in Section 8.3.2).

In the second step, the recommendation of move class destination is correct if and only if it suggests to move the misplaced class back to the package where it was before the smell injection. Notably, all misplaced classes, regardless of the prediction of evaluated smell detection approaches, are employed in the second step to evaluate the accuracy in recommending solutions for misplaced classes. Other classes, including false positives in the preceding step, are excluded from the second step evaluation.

8.3.5 Evaluation of Bootstrap Aggregating

To answer research question *RQ8*, we remove the bootstrap aggregating from the proposed approach and repeat the evaluation processes specified in Section 8.3.1-8.3.4. Without bootstrap aggregating, we train a single classifier for each category of code smells with all of the generated data.

Notably, this time we evaluate the proposed approach only, and none of the baseline approaches is applied because they do not employ bootstrap aggregating.

8.3.6 Performance Metrics

After the evaluation, we calculate the precision, recall, and F_1 of the proposed approach (and the involved baseline approaches as well) in identifying code smells as follows:

$$precision = \frac{true\ positives}{true\ positives\ +\ false\ positives} \quad (31)$$

$$recall = true\ positives \div injected\ smells \quad (32)$$

$$F_1 = 2 \times \frac{precision \times recall}{precision + recall} \quad (33)$$

Besides those metrics, we also employ Area Under Curve (AUC), and Matthews Correlation Coefficient (MCC) for performance comparison where AUC is computed by well-known machine learning tool *scikit-learn* (publicly available at https://scikit-learn.org/dev/index.html) and MCC is computed as follows:

$$MCC = \frac{TP \times TN - FP \times FN}{\sqrt{(TP+FP)(TP+FN)(TN+FP)(TN+FN)}} \quad (34)$$

where TP, TN, FP, and FN are numbers of true positives, true negatives, false positives, and false negatives, respectively.

The accuracy of the approaches in recommending destinations for smelly methods/classes is computed as follows:

$$ac = \frac{correct\ recommendation\ for\ true\ positives}{true\ positives} \quad (35)$$

The recommended destination for misplaced method/class is correct if and only if it suggests to move the method/class back to its enclosing class/package before it is moved. Notably, recommended destinations for false positives are not counted in while computing the accuracy because only if developers confirm that the detected method/class should be moved (i.e., it is a true positive) they should consider the destination problem.

8.4 RQ1:Detection of Feature Envy

To answer research question *RQ1* we compare the proposed approach against *JDeodorant* in detecting feature envy smells. Evaluation results are presented in Table 2. The

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2019.2936376, IEEE Transactions on Software Engineering

TABLE 3
Accuracy in Recommending Target Classes

Applications	Proposed Approach	JDeodorant
Areca	88.42%	62.50%
Freeplane	80.01%	67.57%
jEdit	76.98%	80.00%
JUnit	67.57%	38.46%
PMD	89.19%	90.00%
Weka	78.74%	40.97%
AbdExtractor	63.35%	40.63%
Grinder	69.23%	50.00%
Art of Illusion	70.31%	55.56%
JExcelAPI	77.08%	20.00%
Average	**76.35%**	**49.10%**

first column presents the subject applications. Columns 2-6 presents the precision, recall, F-measure, MCC, and AUC of the proposed approach, respectively. The performance of *JDeodorant* is presented in columns 7-11. The last row of the table presents the average performance of the involved approaches.

From Table 2 we make the following observations:

- First, the proposed approach significantly outperforms the state-of-the-art as synthetic metrics (F-measure, MCC and AUC) are concerned. For example, its average F-measure is 51.91% whereas the average F-measure of *JDeodorant* is 24.51%. Compared to *JDeodorant*, the proposed approach improves F-measure by 27.4% (=51.91%-24.51%). Notably, *JDeodorant* dose not provide thresholds for tuning, and thus its AUC may fail to reveal its real potential.
- Second, the proposed approach can identify most of the feature envy smells. Its average recall is up to 88.11%. Compared to *JDeodorant*, it improves recall dramatically by 71.51% (=88.11%-16.60%).
- Third, *JDeodorant* results higher precision (46.87%) than that of the proposed approach (36.79%) at the cost of the significant reduction in recall.

We conclude from these results in the preceding paragraphs that the proposed approach outperforms the state-of-the-art approaches significantly in detecting feature envy smells.

8.5 RQ2:Recommendation of Target Classes

Methods associated with feature envy smells should be relocated. However, before such methods could be relocated, we have to choose their target classes (destinations). The accuracy of the involved approaches in recommending the target classes is presented in Table 3. Each row of the table presents their accuracy on a subject application except the last row that presents the average accuracy on all subject applications.

From Table 3, we make the following observations:

- First, the proposed approach is accurate in recommending destinations for smelly methods. Its accuracy varies from 63.35% to 88.42%. On average, its accuracy is up to 76.35%. In other words, for around three out of four true positives (i.e., methods that should be moved) the proposed approach can predict correctly to which classes the methods should be moved.
- Second, the proposed approach is more accurate than *JDeodorant* in recommending destinations for smelly methods. Compared to *JDeodorant*, it improves the accuracy by 27.25% (=76.35%-49.10%).

We conclude from these results that the proposed approach is accurate in recommending destinations for feature envy methods, and it improves the state-of-the-art significantly.

8.6 RQ3:Detection of Long Methods

To answer research question *RQ3*, we compare the proposed approach against *DECOR* in detecting long methods. Evaluation results are presented in Table 4. From this Table, we make the following observations:

- First, the proposed approach significantly outperforms the state-of-the-art as synthetic metrics are concerned. Its average F-measure and MCC are 55.53% and 39.53%, respectively. In constrast, the average F-measure and MCC of *DECOR* are 10.42% and 12.30%, respectively. Compared to *DECOR*, the proposed approach improves F-measure and MCC by 15.11%(=55.53%-10.42%), and 27.23% (=39.53%-12.30%), respectively.
- Second, the proposed approach can identify most of the long methods. Its average recall is up to 78.99%. Compared to *DECOR*, the proposed approach improves recall by 73.26%(=78.99%-5.73%).
- Third, *DECOR* improves precision from 42.81% to 57.32% at the cost of significant reduction (by 73.26% in average) in recall.

We conclude from these results in the preceding paragraphs that the proposed approach outperforms the state-of-the-art approach significantly in detecting long methods.

8.7 RQ4:Detection of Large Classes

To answer research question RQ4, we compare the proposed approach against *DECOR*. Evaluation results are presented in Table 5. From this table, we make the following observations:

- First, the proposed approach outperforms *DECOR* concerning synthetic measurements, i.e., F_1 scores, MCC, and AUC. The improvement is 27%=(22.33%-17.6%)/17.6%, 90%=(21.16%-11.16%)/11.16%, and 5%=(75.77%-72.36%)/72.36%, respectively.
- Second, the proposed approach improves recall significantly at the cost of reduced precision. It improves recall from 10.75% to 80.95%, suggesting that most of the large classes could be identified by the proposed approach. However, it also results in low precision (12.95%).

We conclude from the preceding analysis that the proposed approach improves the state of the art in large class detection.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2019.2936376, IEEE Transactions on Software Engineering

TABLE 4
Evaluation Results on Long Method Detection

Applications	Proposed Approach					DECOR				
	$precision$	$recall$	F_1	MCC	AUC	$precision$	$recall$	F_1	MCC	AUC
Areca	42.72%	73.83%	54.13%	39.19%	78.53%	58.10%	6.19%	11.18%	13.50%	77.46%
Freeplane	46.42%	75.61%	57.52%	38.42%	78.79%	63.22%	7.01%	12.61%	14.24%	76.40%
jEdit	52.17%	83.45%	64.20%	40.92%	77.15%	57.38%	4.18%	7.80%	7.66%	74.96%
JUnit	58.53%	52.91%	55.58%	34.27%	72.63%	61.90%	5.42%	9.96%	10.25%	69.84%
PMD	37.09%	70.59%	48.63%	35.18%	77.37%	53.19%	7.35%	12.92%	14.44%	75.55%
Weka	50.23%	79.25%	61.49%	46.00%	81.75%	69.59%	5.38%	9.99%	14.22%	82.01%
AbdExtractor	32.34%	80.63%	46.16%	32.87%	78.81%	50.85%	7.22%	12.64%	13.66%	78.63%
Grinder	37.16%	71.76%	48.96%	29.25%	74.07%	38.89%	4.12%	7.45%	5.77%	69.71%
Art of Illusion	37.60%	87.62%	52.62%	39.32%	80.27%	53.13%	4.54%	8.37%	10.54%	78.69%
JExcelAPI	32.61%	83.63%	46.92%	41.36%	88.53%	32.84%	6.42%	10.76%	9.58%	88.08%
Average	**42.81%**	**78.99%**	**55.53%**	**39.53%**	**79.24%**	**57.32%**	**5.73%**	**10.42%**	**12.30%**	**77.92%**

TABLE 5
Evaluation Results on Large Class Detection

Applications	Proposed Approach					DECOR				
	$precision$	$recall$	F_1	MCC	AUC	$precision$	$recall$	F_1	MCC	AUC
Areca	11.43%	80.00%	20.00%	17.40%	68.82%	75.00%	2.42%	4.69%	3.18%	70.62%
Freeplane	11.97%	70.00%	20.44%	16.83%	72.76%	20.83%	9.09%	12.66%	3.20%	72.99%
jEdit	15.00%	75.00%	25.00%	23.42%	77.39%	58.82%	15.63%	24.69%	15.04%	75.28%
JUnit	11.76%	40.00%	18.18%	9.76%	71.75%	40.00%	18.18%	25.00%	19.02%	71.50%
PMD	16.67%	100%	28.57%	31.71%	83.49%	100%	17.50%	29.79%	33.89%	81.77%
Weka	10.06%	94.44%	18.18%	17.45%	68.55%	18.75%	4.48%	7.23%	-1.35%	60.97%
AbdExtractor	16.46%	79.41%	27.27%	27.04%	79.75%	70.97%	13.10%	22.11%	17.27%	78.12%
Grinder	12.73%	70.00%	21.54%	18.67%	79.15%	50.00%	24.14%	32.56%	26.29%	64.32%
Art of Illusion	12.33%	81.82%	21.43%	20.13%	78.49%	0%	0%	0%	-4.24%	55.41%
JExcelAPI	22.00%	84.62%	34.92%	36.26%	80.89%	35.29%	18.18%	24.00%	16.40%	81.21%
Average	**12.95%**	**80.95%**	**22.33%**	**21.16%**	**75.77%**	**48.39%**	**10.75%**	**17.60%**	**11.16%**	**72.36%**

8.8 RQ5:Detection of Misplaced Classes

To answer research question RQ5, we compare the proposed approach against *TACO* in the identification of misplaced classes. Evaluation results are presented in Table 6. From this table, we make the following observations:

- First, the proposed approach outperforms *TACO* concerning synthetic metrics, i.e., F_1, MCC, and AUC on all of the subject applications. The improvement in both F_1 (34%=(87.34%-65.23%)/65.23%) and MCC (49%=(84.03%-56.44%)/56.44%) is significant.
- Second, the proposed approach improves both precision and recall significantly. The improvement in precision is 16.8%=(81.6%-64.8%) whereas the improvement in recall is 28.3%=93.96%-65.66%.

We conclude from the preceding analysis that the proposed approach outperforms the baseline in misplaced class detection.

8.9 RQ6:Recommendation of Target Packages

To answer research question RQ6, we compare the proposed approach again *TACO* in recommendation of target packages for misplaced classes. The accuracy of the recommendation is critical because if the misplaced classes are moved to wrong positions, they remain *misplaced*.

Evaluation results are presented in Table 7. The first column presents subject applications. The second and the fourth columns present the numbers of accepted target packages recommended by the proposed approach and *TACO*, respectively. The third and fifth columns present the accuracy of the proposed approach and *TACO*, respectively. From this table, we make the following observations:

- First, the proposed approach results in greater number of accepted recommendations. In total, 488 of its recommendations are accepted whereas the number is reduced to 342 for *TACO*.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2019.2936376, IEEE Transactions on Software Engineering

TABLE 6
Evaluation Results on Misplaced Class Detection

Applications	Proposed Approach					TACO				
	precision	*recall*	F_1	*MCC*	*AUC*	*precision*	*recall*	F_1	*MCC*	*AUC*
Areca	93.62%	92.63%	93.12%	91.46%	99.78%	45.63%	53.41%	49.21%	35.26%	81.66%
Freeplane	94.94%	91.46%	93.17%	91.41%	99.77%	53.92%	68.75%	60.44%	51.72%	88.12%
jEdit	55.04%	100.00%	71.00%	65.39%	99.18%	67.27%	66.07%	66.67%	58.75%	92.13%
JUnit	41.38%	100.00%	58.54%	51.55%	99.47%	38.24%	68.42%	49.06%	36.39%	46.15%
PMD	56.10%	97.87%	71.32%	67.36%	99.57%	62.50%	62.50%	62.50%	52.88%	89.94%
Weka	96.83%	94.18%	95.49%	94.28%	99.83%	86.70%	75.12%	80.49%	75.77%	95.11%
AbdExtractor	95.08%	92.28%	93.66%	92.03%	99.68%	68.45%	63.68%	65.98%	57.48%	89.33%
Grinder	92.50%	82.22%	87.06%	84.68%	98.43%	42.59%	62.16%	50.55%	43.21%	89.43%
Art of Illusion	76.42%	100.00%	86.64%	82.42%	99.44%	92.68%	54.29%	68.47%	61.78%	94.51%
JExcelAPI	39.58%	90.48%	55.07%	46.39%	92.49%	41.18%	73.68%	52.83%	47.81%	35.71%
Average	**81.60%**	**93.96%**	**87.34%**	**84.03%**	**99.09%**	**64.80%**	**65.66%**	**65.23%**	**56.44%**	**89.73%**

TABLE 7
Accuracy in Recommending Target Packages

Applications	Proposed Approach		TACO	
	Correct	Accuracy	Correct	Accuracy
Areca	38	43.18%	27	57.45%
Freeplane	43	57.33%	33	60.00%
jEdit	32	45.07%	17	45.95%
JUnit	13	54.17%	6	46.15%
PMD	27	58.70%	16	64.00%
Weka	172	62.55%	136	83.44%
AbdExtractor	87	34.66%	66	51.56%
Grinder	20	54.05%	10	43.48%
Art of Illusion	49	52.13%	26	68.42%
JExcelAPI	7	36.84%	5	35.71%
Total	**488**	**49.80%**	**342**	**62.98%**

- Second, *TACO* is more accurate than the proposed approach in recommending target packages. It improves the average accuracy from 49.80% to 62.98%. However, on the same code smells where both the proposed approach and *TACO* make recommendations, their accuracy in recommending target packages is close to each other: 62.6% for *TACO* and 60.5% for the proposed approach.

Based on the preceding analysis, we conclude that the proposed approach outperforms the baseline in identifying misplaced classes, and it is comparable to the baseline in recommending target packages.

8.10 RQ7: Efficiency of the Proposed Approach

To investigate the efficiency (time complexity) of the neural network based classifier, we record the time spent on training and testing during evaluation. To improve the efficiency of training, we conduct the training on a workstation with GPU whose setting is listed as follows: 64GB RAM, Intel Xeon CPU E5-2660 v4 2.00GHz, NVIDIA Quadro M4000. In contrast, the testing is conducted on a personal computer without GPU:16GB RAM, Intel Core i7-6700 CPU 3.40GHz. We conduct the testing on personal computer instead of more powerful workstation because detection of feature envy (testing) is often conducted on personal computers by developers.

Evaluation results suggest that the proposed approach is efficient. On average, the training could be done within 9 minutes for long method detection, 7 minutes for large class detection, 18 minutes for feature envy detection, and 12 minutes for misplaced class detection. With the trained classifier, it takes 0.2-20.7 minutes (8 minutes on average) to detect long method smells for one subject application. For the detection of large classes, feature envy detection, and misplaced classes, it takes the trained classifiers 0.44, 5.3, and 8.44 minute on average to check one subject application. One possible reason for the high efficiency of the proposed deep learning based approach is that the employed neural networks are much smaller than those widely employed in image or natural language processing, e.g., Google translator. For example, the network we employ for feature envy detection has only six layers and the maximal layer contains 128 neurons only. As a result, the training of such networks could be done much more quickly than complex deep neural networks that are composed of millions of neurons. Another reason for the high efficiency is that the training data are rather small. For each kind of the involved code smells, we generate no more than 100 thousand items for training because the neural networks to be trained are rather small.

Further analysis on the testing time suggests that most (99.95% for long method detection, 75.13% for large class detection, 87.18% for feature envy detection, and 95.81% for misplaced class detection) of the testing time is spent on information extraction, i.e., extracting metrics and textual information. In contrast, the neural network always makes predictions instantly, and it consumes less than 25% of the

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2019.2936376, IEEE Transactions on Software Engineering

testing time.

8.11 RQ8: Effect of Bootstrap Aggregating

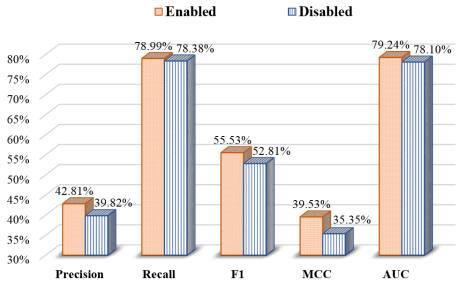


Fig. 8. Effect of Bootstrap Aggregating on Long Method Detection

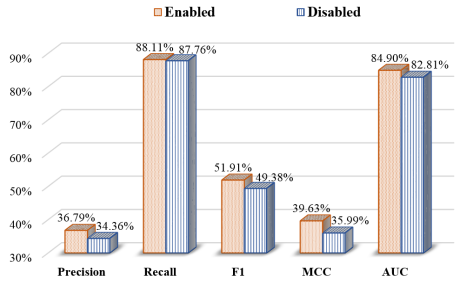


Fig. 9. Effect of Bootstrap Aggregating on Feature Envy Detection

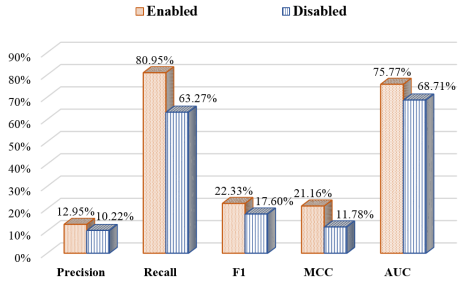


Fig. 10. Effect of Bootstrap Aggregating on Large Class Detection

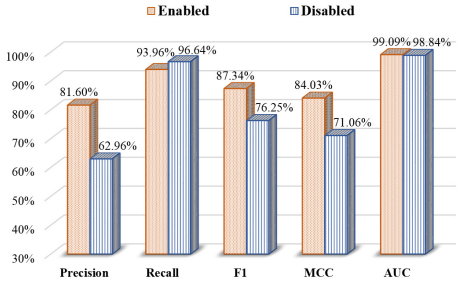


Fig. 11. Effect of Bootstrap Aggregating on Misplaced Class Detection

To investigate how the bootstrap aggregating influences the performance of the proposed approach, we evaluate the proposed approach with and without bootstrap aggregating, respectively. Evaluation results are presented in Fig. 8-Fig. 11. From the figures, we make the following observations:

- Bootstrap aggregating improves the overall performance of the proposed approach by increasing F_1 from 52.8% to 55.53% in long method detection, from 49.38% to 51.94% in feature envy detection, from 17.6% to 22.33% in large class detection, and from 76.25% to 87.34% in misplaced class detection.
- Bootstrap aggregating has significant positive impact on precision. It improves precision from 39.82% to 42.81% in long method detection, from 34.36% to 36.79% in feature envy detection, from 10.22% to 12.95% in large class detection, and from 62.96% to 81.60% in misplaced class detection. With the bootstrap aggregating, the proposed approach suggests smells only if multiple classifiers make the same suggestion. Consequently, it makes fewer risky suggestions, which in turn improves the precision.
- The bootstrap aggregating has minor (and inconsistent) impact on recall. It slightly improves recall from 78.38% to 78.99% in long method detection, and from 87.76% to 88.11% in feature envy detection. Although the improvement in large class detection is more significant (from 63.27% to 80.95%), it results in reduction in recall of misplaced class detection (from 96.64% to 93.96%).

Based on the preceding analysis, we conclude that bootstrap aggregating helps to improve the performance, especially precision, of the proposed approach.

8.12 Threats to Validity

The first threat to validity is that the code smells (feature envy, long methods, large classes, and misplaced classes) involved in the evaluation are generated automatically. It is likely that such automatically generated smells are different from those that are manually introduced by developers during real developments. Consequently, conclusions drawn on such generated data set may not hold on real applications. To reduce the threat, while creating feature envy smells/misplace classes, we randomly select methods/classes to move, and randomly select the target classes/pakcages for such methods/classes. We also rigidly control the ratio of smelly methods/class to make sure that the ratio is close to that in real applications. Finally, we carry out a case study in Section 9 where no automatically generated smells are involved. The results of the case study confirm the conclusion that the proposed approach improves the state-of-the-art.

The second threat to validity is that the evaluation is based on the assumption that the involved software entities in the subject applications are well designed. However, the assumption may not hold, and thus the calculation of performance (e.g., precision and recall) based on this assumption may be inaccurate. To reduce the threat, we only select well-known high quality subject applications for the evaluation.

The third threat to validity is that only ten subject applications are involved in the evaluation. Special characters of such applications may bias the conclusions, and thus such conclusions may not hold on other applications. To reduce the threat, we select applications from different domains and different developers. We also carry out leave-one-out validation on such applications to reduce the bias introduced by specific applications. To facilitate further evaluation, we

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2019.2936376, IEEE Transactions on Software Engineering

publish the source code of the implementation as well as the evaluation data [64].

The fourth threat to validity is that during the evaluation we re-implement baseline approach TACO and modify another baseline approach JDeodorant. TACO is not publicly available, and thus we re-implement it for the evaluation. Although JDeodorant is open-source, we modify it because we need a special version of it (disabling some pre-conditions). However, the re-implementation and modification may be incorrect, which could significantly influence the evaluation results. To reduce the threats, we recruit experienced code reviewers to double check the implementation/modification. We also compare the performance of the re-implementation of TACO against that reported by the authors [42], and the comparison suggests that they are comparable. Finally, we make them publicly available [64] so that other researchers may further validate them in future.

Finally, we compare the proposed approach with structural-based techniques only (except for the misplaced class detection). The reason is that the metrics used by the proposed technique are all structural, and thus a comparison with similar approaches can give insights on the validity of the deep learner. However, other types of information (e.g., textual or historical analysis) can actually complement the identification power of structural-based methods. It would be interesting to compare the proposed approach with such code smell detectors based on different independent variables.

A special baseline is random guess: If the ratio of smelly software entities in sample applications is r_0, random guess would predict software entities in testing project as positive (smelly) at the chance of r_0. However, since r_0 is often small (e.g., $r_0 = 2.3\%$ for feature envy), random guess would lead to extremely low recall (the average recall is equal to r_0).

9 Case Study

In Section 8, we evaluate the proposed approach on applications with automatically injected smells. In this section, we evaluate it with real applications without any injection.

9.1 Research Questions

This section investigates the following research questions:

- **CS-RQ1**: Does the proposed approach outperform the state-of-the-art approaches on real-world projects?
- **CS-RQ2**: What are the major characters of the generated training data? Can we exclude some *low-quality* data with pre-defined quality metrics, and how will the exclusion influence the performance of the proposed approach?

Research question CS-RQ1 investigates the performance of the proposed approach in identifying real-world code smells compared against state-of-the-art baseline approaches introduced in Section 8. The automatically generated testing data employed in the evaluation in Section 8 may be different from real-world code smells. Consequently, in this section we further evaluate the proposed approach on real-world projects.

Research question CS-RQ2 investigates the quality of generated training data and potential preprocessing of these data. For example, how often a generated feature envy method accesses more methods/abbributes (noted as T) from its target class than those (noted as S) from its enclosing class? Should we exclude such feature envy methods where $|T| > |S|$ dose not hold?

TABLE 8
Subject Applications for Case Study

Applications	Domain	Version	NOC	NOM	LOC
XMD	Download Manager	2.1	198	1599	42,604
JSmooth	Java Wrapper	0.9.9	91	669	16,782
Neuroph	Neural Network	2.9	214	1186	26,513
DavMail	Email	4.5.1	137	1,178	31,915
CKEditor	Editor	2.6	40	224	4,030

9.2 Subject Applications

We search for subject applications from SourceForge as follows. First, we search for Java applications. Second, we sort the resulting applications by popularity in ascending order assuming that the less popular applications may contain more code smells. Third, we exclude such applications whose LOC (lines of source code) is smaller than four thousand or greater than fifty thousand. If the subject application is too large, it may take a long time to manually check the potential code smells reported by the involved approaches. In contrast, if the subject application is too small, the number of code smells may be small (or no smells at all), which may reduce the statistical significance of the evaluation results. Consequently, we limit the size of the subject application to a given range. Fourth, we exclude those applications that cannot be compiled successfully. Syntactical errors may prevent the proposed approach (and the baseline approaches as well) to extract necessary information correctly. Finally, we select the top five of the remaining subject applications. Table 8 presents the information of such applications.

XDM [85] (Xtreme Download Manager) is a download manager to save streaming videos from websites, resume broken/dead downloads, and schedule downloads. *JSmooth* [86] is a Java executable wrapper that creates native Windows launchers for given Java applications. *Neuroph* [87] is a lightweight neural network framework providing Java neural network library and GUI tool to facilitate creating, training and saving of neural networks. *DavMail* [88] is an exchange gateway allowing users to use any mail/calendar client with an Exchange server. It allows users to interact with Exchange servers behind a firewall. *CKEditor* [89] is modern rich text editor with a modular architecture.

9.3 Process

9.3.1 Comparison against Existing Approaches

For each of the subject applications, the evaluation on feature envy detection is as follows. First, we train the proposed deep learning based feature envy detection classifier with data generated from the ten subject applications that

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2019.2936376, IEEE Transactions on Software Engineering

TABLE 9
Results of Case Study on Feature Envy Detection

Metrics	Proposed Approach						JDeodorant					
	XMD	JSmooth	Neuroph	DavMail	CKEditor	**Total**	XMD	JSmooth	Neuroph	DavMail	CKEditor	**Total**
#Report	26	13	77	23	2	**141**	18	3	19	32	0	**72**
#Accepted	12	5	30	8	1	**56**	6	1	5	9	0	**21**
#Accepted targets	9	4	25	6	1	**45**	2	1	3	4	0	**10**
Precision	46%	38%	39%	35%	50%	**40%**	33%	33%	26%	28%	Nan	**29%**
Accuracy (destination)	75%	80%	83%	75%	100%	**80%**	33%	100%	60%	44%	Nan	**48%**

are introduced in Table 1. Second, we apply the proposed approach (trained classifier) and *JDeodorant* to the selected subject application independently, and merge the detection results. Third, we present the detection results to three developers for manual checking. Fourth, based on the manual checking, we compute the performance of the evaluated approaches.

The manual checking is accomplished by three post-graduate students in Beijing Institute of Technology. All of them are majored in computer science and have rich experience in Java development. They check the detection results independently. After that, they discuss together to remove inconsistence. Notably, the evaluation is conducted in an anonymous manner, and thus the students cannot identify the tool that performed the recommendation.

The evaluation on the detection of other code smells (i.e., long method, large class, and misplaced class) is similar to that on feature envy detection. The reported potential code smells are manually checked by the same participants in the same way as feature envy is checked. The only difference is that we replace the evaluated feature envy detection approaches with corresponding detection approaches.

9.3.2 Quality of Generated Data and Its Influence

For each method (m) that are employed to generate positive or negative feature envy item (as specified in Section 9.3.1), we compute $\Delta(m) = |S(m)| - |T(m)|$ where $|S(m)|$ represents the number of methods/abbributes from the enclosing class of m that are accessed by m. $|T(m)|$ is the maximal number of methods/abbributes from an outside class (other than the enclosing class) that are accessed by m. $\Delta(m)$ is frequently employed to identify feature envy smells based on the assumption (noted as A_{fe}) that a well-placed method should access more methods/attributes from its enclosing class than any of its potential target classes [39]. Consequently, drawing the distribution of $\Delta(m)$ for involved methods may reveal how often such methods are *well-placed*, and how often the generated feature envy smells conform to the assumption A_{fe}. We also exclude such generated data that do not conform to the assumption A_{fe} (we call them *odd* data for short), and repeat the evaluation in Section 9.3.1. By comparing the evaluation results before and after the exclusion, we may reveal how the data preprocess (i.e., excluding *odd* training data) influences the performance of the proposed approach.

In a similar way, we evaluate the quality of generated long methods, large classes, and misplaced classes. For a positive long method created by inlining method im into another method m, we compute $\Delta(m, im) = COH(m') - COH(m)$ where $COH(m)$ and $COH(m')$ are the cohesion of method m before and after method im is inlined into m. We exclude *odd* positive items whose $\Delta(m, im) \geq 0$, and repeat the evaluation; For a large class mc created by merging classes c_1 and c_2, we compute $\Delta(c_1, c_2) = TCC(mc) - average(TCC(c_1), TCC(c_2))$ where $TCC(mc)$ is the Tight Class Cohesion of the generated large class; For each class (c) that are employed to generate positive or negative misplaced class item, we compute $\Delta(c) = sumMPC(c, ep) - sumMPC(e, tp)$ where $sumMPC(c, ep)$ is Message Passing Coupling between c and its enclosing class. $sumMPC(e, tp)$ is the maximal Message Passing Coupling between c and one of its outside packages (other than the enclosing package ep).

9.4 CS-RQ1: Improving The State of The Art

Table 9 presents the results on feature envy detection. The first column presents different metrics, like precision and accuracy. Columns 2-7 presents the results of the proposed approach on different applications, and the seventh column presents its average performance on all applications. The performance of *JDeodorant* is presented in columns 8-13. The second row presents the number of potential feature envy reported by different approaches, and the third row presents the number of accepted (correct) feature envy. The fourth row presents the number of correct recommendation of destinations for the accepted feature envy. The last two rows present the precision in detecting feature envy and accuracy in recommending destinations, respectively. From Table 9, we make the following observations:

- First, the proposed approach can identify feature envy smells in real applications, and suggest the correct target classes for smelly methods. In total, it successfully detect 141 feature envy smells from the five subject applications, among which 56 are manually confirmed. It also successfully recommends the correct destinations for 80% of the confirmed smelly methods.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2019.2936376, IEEE Transactions on Software Engineering

Anova: Single Factor

SUMMARY

Groups	*Count*	*Sum*	*Average*	*Variance*
Column 1	5	56	11.2	126.7
Column 2	5	21	4.2	13.7

ANOVA

Source of Variation	*SS*	*df*	*MS*	*F*	*P-value*	*F crit*
Between Groups	122.5	1	122.5	1.745014	0.223037	5.317655
Within Groups	561.6	8	70.2			
Total	684.1	9				

Fig. 12. ANOVA Analysis on Numbers of Accepted Feature Envy

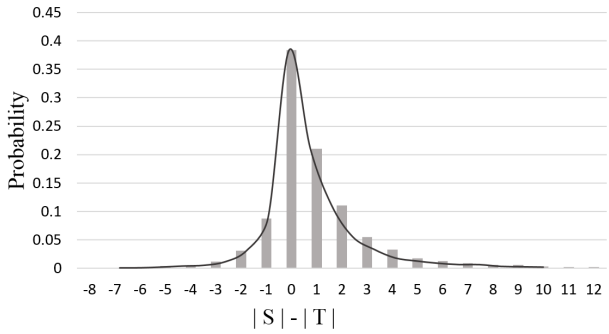


Fig. 13. Distribution of $|S| - |T|$ for Seed Methods (Feature Envy)

- Second, the proposed approach significantly outperforms *JDeodorant* in detecting feature envy smells. It improves the precision from 29% significantly to 40%. It also identifies much more true positives than *JDeodorant* (56 VS. 21), which suggests that the proposed approach achieves greater recall than *JDeodorant*.
- Third, the proposed approach is more accurate than *JDeodorant* in recommending destinations for feature envy methods. Compared to *JDeodorant*, it improves the accuracy by 32% (=80%-48%).

We conduct Analysis Of Variance (ANOVA) on the resulting performance. Fig. 12 presents the ANOVA results on the numbers of accepted feature envy smells by the involved approaches. From this figure we observe that the p-value (0.223) is significantly greater than 0.05 although the proposed approach identifies more feature envy smells than *JDeodorant* in four out of the five subject applications (as shown in Table 9). One possible reason is that the number varies dramatically from project to project even for the same approach (e.g., the proposed approach). On the other side, the effect size (Cliff's Delta) on the same numbers is 0.44, suggesting that the proposed approach significantly improves the number of accepted feature envy smells.

Table 10 presents the evaluation results on the detection of misplaced classes. The structure of the table is highly similar to that of Table 9. From this table, we make the following observations:

- First, the proposed approach successfully identifies more real smells. On the five projects, it identifies 152 manually confirmed misplaces classes whereas *TACO* identifies 27 only.
- Second, the proposed approach results in much more accepted target packages (77) than that of *TACO* (16).
- Third, the proposed approach results in higher precision in misplaced class detection than *TACO*. The improvement is up to 16%=70%-54%.
- Finally, the precision of *TACO* in recommending target packages for misplaced classes is slightly higher than that of the proposed approach. It improves the accuracy from 51% to 59%.

Table 11 and Table 12 present the results on long method detection and large class detection, respectively. From these tables, we make the following observations:

- First, the proposed approach identifies more true positives than *DECOR*. As suggested by Table 11, on each of subject applications the proposed approach identifies more manually confirmed long methods. In total, it successfully identifies 581 real long methods. In contrast, *DECOR* identities 64 only. The same is true on the identification of large classes: the proposed approach identifies 45 manually confirmed large classes whereas *DECOR* identifies 8.
- Second, the precision of the proposed approach is smaller than that of *DECOR* in identification of large classes and long methods. One of the possible reasons is that *DECOR* tends to make few suggestions, which results in greater precision at the cost of reduced recall. We notice that compared to *DECOR*, the proposed approach achieves significant improvement in recall (809%=(581-64)/64 in long method detection and 463%=(45-8)/8 in large class detection) at the cost of a relatively smaller reduction in precision 5.3%=(57%-54%)/57% in long method detection and 20%=(15%-12%)/15% in large class detection). Notably, by comparing the number of true positives (tp) of different approaches, we can know how the recall has changed because the total number of real smells is a constant for the case study and it is independent of detection approaches.
- The effect size on the accepted numbers of long methods and large classes is 1 and 0.84, respectively. The effect size suggests that compared to *DECOR* the proposed approach does significantly improve the accepted numbers of long methods and large classes.

We conclude from the preceding analysis that the proposed approach significantly outperform the state-of-the-art in detecting. This is consistent with the evaluation results presented in Section 8.

9.5 CS-RQ2:Quality of Generated Data and Influence of Filtering

9.5.1 Feature Envy

For methods that are employed to create positive or negative feature envy items (we call them *seed methods* for short), we present the distribution of their $\Delta(m) = |S(m)| - |T(m)|$ in Fig. 13. From this figure, we make the following observations:

- First, $|S(m)| - |T(m)| > 0$ holds on around half of the seed methods (47%). Training data generated based on such seed methods conform to the assumption A_{fe}, i.e., a well-placed method should access more

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2019.2936376, IEEE Transactions on Software Engineering

TABLE 10
Results of Case Study on Misplaced Class Detection

Metrics	Proposed Approach						TACO					
	XMD	JSmooth	Neuroph	DavMail	CKEditor	**Total**	XMD	JSmooth	Neuroph	DavMail	CKEditor	**Total**
#Report	44	9	151	6	7	**217**	16	2	17	9	6	**50**
#Accepted	32	7	105	4	4	**152**	9	0	10	4	4	**27**
#Accepted targets	17	3	51	2	3	**77**	6	0	5	2	3	**16**
Precision	73%	78%	70%	67%	57%	**70%**	56%	0%	59%	44%	67%	**54%**
Accuracy (destination)	53%	43%	49%	50%	75%	**51%**	67%	Nan	50%	50%	75%	**59%**

TABLE 11
Results of Case Study on Long Method Detection

Metrics	Proposed Approach						DECOR					
	XMD	JSmooth	Neuroph	DavMail	CKEditor	**Total**	XMD	JSmooth	Neuroph	DavMail	CKEditor	**Total**
#Report	429	165	487	420	24	**1525**	28	13	50	16	6	**113**
#Accepted	179	65	171	153	13	**581**	16	8	27	9	4	**64**
Precision	42%	39%	35%	36%	54%	**38%**	57%	62%	54%	56%	67%	**57%**

TABLE 12
Results of Case Study on Large Class Detection

Metrics	Proposed Approach						DECOR					
	XMD	JSmooth	Neuroph	DavMail	CKEditor	**Total**	XMD	JSmooth	Neuroph	DavMail	CKEditor	**Total**
#Report	114	68	120	65	19	**386**	13	7	11	19	3	**53**
#Accepted	10	7	15	9	4	**45**	1	1	2	3	1	**8**
Precision	9%	10%	13%	14%	21%	**12%**	8%	14%	18%	16%	33%	**15%**

methods/attributes from its enclosing class than any of its potential target classes whereas a feature envy method should access fewer methods/attributes from its enclosing class than at least one of its potential target classes.

- Second, $|S(m)| - |T(m)| < 0$ holds on only a small part of the seed methods (14%). Training data generated based on such methods will not conform to the assumption A_{fe}.
- Third, for a significant part (38%) of the seed methods, $|S(m)| = |T(m)|$ holds. Training data generated based on such methods are on the bordering of the assumption A_{fe}.

We exclude such seed methods where $|S(m)|-|T(m)| \leq 0$, and repeat the evaluation in Section 9.3.1. Evaluation results suggest that the number of accepted feature envy smells decreases significantly from 56 (default setting) to 36 whereas the precision decreases slightly from 39.72% to 35.29%. One possible reason for the reduction in performance is that filtering generated data according to predefined assumptions (e.g., assumption A_{fe}) may result in overfitting to the assumptions. However, the assumptions may not hold in some cases, and thus overfitting to them may reduce the generalizability of the resulting classifiers. An illustrating example from open-source project *Areca* is presented in Fig. 14. Although the method *buildEntriesMap* accesses more (7) methods/attributes from class *ArecaRawFileList* than those (0) from its enclosing class *AbstractIncrementalFileSystemMedium*, it is a well-placed method because its functionality (building entry map of archives for recovery) is beyond the scope of the target class *ArecaRawFileList* that is essentially a data structure to store a list of raw files. In contrast, its functionality falls in the scope of its enclosing class *AbstractIncrementalFileSystemMedium* that is in charge of file systems. Consequently, although this method does not conform to the assumption A_{fe}, it is a qualified seed method for data generation, and excluding it from training data generation may not help.

9.5.2 Long Methods

Fig. 15 presents the probability distribution of $COH(m') - COH(m)$ for seed methods that are employed to create positive long method items. From this figure, we observe that most (88%) of the seed methods become less cohesive after inline method refactorings.

We exclude such seed methods that become more cohesive after inline method refactorings, and repeat the evaluation in Section 9.3.1. Evaluation results suggest that the

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2019.2936376, IEEE Transactions on Software Engineering


```
    private RecoveryFilterMap buildEntriesMap(File[]
optimizedArchives, ArecaRawFileList filters, File traceFile,
ProcessContext context)
    throws IOException, FileMetaDataSerializationException,
TaskCancelledException, ApplicationException {
    Logger.defaultLogger().fine("Building entries map for " +
optimizedArchives.length + " archives.");

    // Create a normalized copy of the file list
    filters.sort();
    filters.deduplicate();
    ArecaRawFileList normalized =
(ArecaRawFileList)filters.duplicate();
    normalized.removeTrailingSlashes();

    // Dispatch entries
    EntriesDispatcher dispatcher =
this.handler.buildEntriesDispatcher(optimizedArchives);

    try {
      if (! filters.hasDirs()) {
        // Only entries of "file" type -> explicitly dispatch
entries
        for (int e=0; e<normalized.length(); e++) {
          dispatcher.dispatchEntry(normalized.get(e));
        }
      } else {
        // Both files and directories -> traverse trace file
        try {
          EntrySetTraceHandler handler = new
EntrySetTraceHandler(normalized, dispatcher);
          ArchiveTraceAdapter.traverseTraceFile(handler,
traceFile, context);
        } catch (IllegalStateException e) {
          Logger.defaultLogger().fine("Error reading " +
traceFile);
          return null;
        }
      }
    } finally {
      dispatcher.close();
    }

    RecoveryFilterMap entriesByArchive =
dispatcher.getResult();
    Logger.defaultLogger().info("" +
dispatcher.getEntriesCount() + " files will be recovered.");
    return entriesByArchive;
  }
```

Fig. 14. *Odd* But Qualified Seed Method for Data Generation

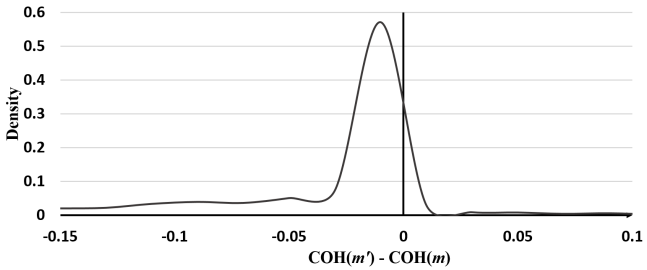


Fig. 15. Distribution of $COH(m') - COH(m)$ for Seed Methods (Positive Long Method Items)

number of accepted long method smells decreases slightly from 581 (default setting) to 569 whereas the precision keeps stable (38%). One possible reason for the minor reduction is that only a small number (12%) of the seed methods are excluded.

9.5.3 Misplaced Classes

Fig. 16 presents the probability distribution of $sumMPC(c, ep) - sumMPC(e, tp)$ for seed classes that are employed to create positive/nagative misplaced class items. From this figure, we observe that a large

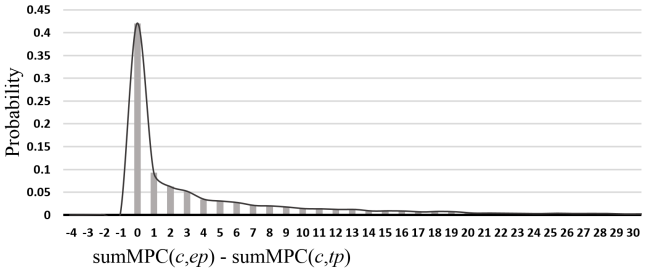


Fig. 16. Distribution of $sumMPC(c, ep) - sumMPC(e, tp)$ for Seed Classes (Misplaced Classes)

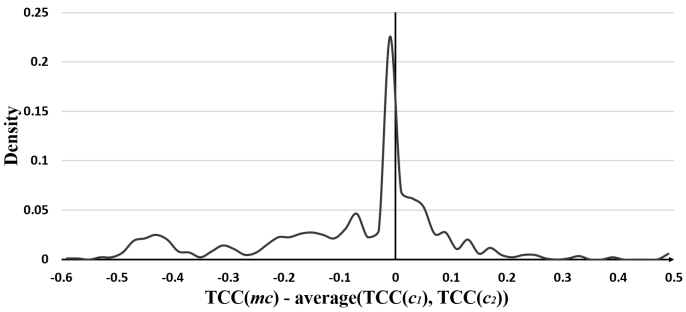


Fig. 17. Distribution of $TCC(mc) - average(TCC(c_1), TCC(c_2))$ for Generated Large Classes

percentage (57.3%) of the seed classes have more message exchange with their enclosing packages than any of their potential target packages. Only a small number (0.7%) of them are coupled more intensively to some target classes, i.e., $sumMPC(c, ep) < sumMPC(e, tp)$. However, we also observe that $sumMPC(c, ep) = sumMPC(e, tp)$ holds on a large number (42%) of the seed classes. One of the major reasons for the equivalence is that the classes often have no message exchange with related packages, i.e., $sumMPC(c, ep) = sumMPC(e, tp) = 0$.

We exclude odd seed classes (where $sumMPC(c, ep) \leq sumMPC(e, tp)$) from training data generation, and repeat the evaluation in Section 9.3.1. Evaluation results suggest that the number of accepted misplaced class smells decreases slightly from 152 (default setting) to 145 whereas the precision decreases slightly from 70% to 67%.

9.5.4 Large Classes

Fig. 17 presents the probability distribution of $\Delta(c_1, c_2) = TCC(mc) - average(TCC(c_1), TCC(c_2))$ for generated large classes. From this figure, we observe that in most cases (68%) merging two classes c_1 and c_2 results in a less cohesive large class.

We exclude such generated large classes where $TCC(mc) \geq average(TCC(c_1), TCC(c_2))$, and repeat the evaluation in Section 9.3.1. Evaluation results suggest that the number of accepted large class smells decreases significantly from 45 (default setting) to 32 whereas the precision increases from 11.7% to 17.2%.

9.5.5 Conclusions

Based on the preceding analysis in Sections 9.5.2-9.5.4, we conclude that filtering out *odd* training data may result in

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2019.2936376, IEEE Transactions on Software Engineering

reduced number of successfully identified code smells, i.e., reduced generalizability of the resulting classifiers.

One possible reason for the performance reduction is that the filtering algorithms (based on some common and straightforward code metrics) employed in Sections 9.5.2-9.5.4 deserve significant improvement. More advanced filtering algorithms may help to improve the quality of generated training data whereas the generalizability of the resulting classifiers trained on such data is not significantly reduced. In future, it will be interesting to design such effective filtering algorithms for the generated training data.

9.6 Threats to Validity

The first threat to validity is that only five subject applications are involved in the case study. Special characters of such applications may bias the conclusions, and thus such conclusions may not hold on other applications. To reduce the threat, we select applications from different domains and different developers. Although additional subject applications would significantly increase the cost (manual checking), it would be interesting in future to evaluate the approaches with more applications.

The second threat to validity is that manual checking of the reported potential code smells could be inaccurate. The three participants are not the original developers of the subject applications, and they may not fully understand the design. Consequently, they may make incorrect judgements on the potential smells. To reduce the threat, we select three participants who have rich experience in software refactoring and Java development. Besides that, we also ask them to discuss together and reach an agreement on all smells. Finally, we also conduct an evaluation in Section 8 where manual checking is not required.

10 Discussion

10.1 Evaluation with Original Developers

The case study in Section 9 recruits external developers to manually check the potential code smells reported by smell detection tools. Such external developers are not familiar with the subject applications, and thus may make mistakes during the manual checking. If the original developers of such applications could be recruited, it would significantly improve the accuracy of the manual checking because they have much better understanding of the subject applications. However, it is challenging to recruit the original developers for manual checking. One of the reasons is that the checking is time consuming and tedious. Another possible reason is that resolving code smells often have low priority in companies.

An alternative way to recruit original developers in the evaluation is to integrate the implementation of the proposed approach into IDEs, and collect feedbacks (i.e., which reported items are confirmed or rejected) automatically by such IDEs. A similar approach has been proposed by Liu et al. [13] to collect manual feedbacks in smell detection tools automatically. They collect such feedbacks to optimize the setting of smell detection tools. Such feedbacks can also be employed to estimate the performance, especially precision, of smell detection tools.

10.2 Generality of the Generated Data

The proposed approach generates labeled training data automatically by randomly selected smell-introducing refactorings. Such generated data serve as the basis of deep learning based code smell detection. However, they may be different from real code smells introduced by developers. As a result, the classifier trained with such generated data may learn to distinguish the randomly refactored entities instead of those that are associated with manually introduced smells. To reduce the threat, we evaluate the proposed approach with a case study in Section 9 where the detected code smells are manually confirmed. The evaluation results suggest that the proposed approach can identify real code smells accurately even if it is trained with generated data. In future, we would investigate how to improve the approach by adding real code smells to the generated training data. Besides that, considering multiple refactoring tools in future during the data generation may help to improve the generality as well.

10.3 Differences between Generated Data and Real-World Data

By comparing the evaluation results in Section 8 and results of case study in Section 9, we observe some differences in the results between the evaluation based on generated data (Section 8) and the evaluation based on manually validated data (Section 9). For example, the precision of the proposed approach in identifying misplaced classes decreases significantly from 81.6% (on generated data) to 70% (on manually validated data). Baseline approaches also suffer similar changes. For example, the precision of *DECOR* in identifying large classes decreases from 48% (on generated data) to 15% (on manually validated data). However, on other cases, the two data sets lead to comparable results. For example, the precision of the proposed approach in feature envy detection changes slightly from 36.79% to 40%, and the precision of *DECOR* in long method detection keeps stable (57% on both generated data and validated data).

One possible reason for the difference in the evaluation results is that generated data could be significantly different from real-world code smells. For example, we may move a well-placed method m from class a to class b to generate a positive feature envy smell instance. However, real-world developers may not ever create such a smell instance by putting m in class b if it *obviously* belongs to a. As a result, the generated smell instance is essentially different from real-world ones that are often more challenging to identify. The difference may suggest that evaluation of code smell detection approaches/tools should rely more on manually validated testing data that are often more reliable than generated data. It may also suggest that the generated data are more valuable as training data than as testing data because advanced learning techniques like deep learning work well even if the training data contain some noise [60].

Another possible reason for the difference is that the performance of the involved approaches varies significantly from case to case regardless of the type of testing data (generated or manually validated). For example, from Table 4 we observe that the precision of long method detection varies significantly from project to project although all the

involved data are generated automatically. It varies from 32.34% to 58.53% (the proposed approach) and from 32.84% to 69.59% (*DECOR*). On the results of the case study in Section 9, we also observe similar things when all testing data are manually validated. For example, from Table 12 we observe that the precision on large class detection varies significantly from 9% to 21% (the proposed approach) and from 8% to 33% (*DECOR*).

10.4 Misplaced Fields

In this paper, we follow a rather narrow definition of feature envy and thus the proposed approach is designed to detect misplaced methods only. In a broad definition, however, feature envy also covers misplaced fields. We follow the narrow definition because it is well recognized [6] and most of the existing code smell detection tools follow this definition. However, it is interesting to extend the proposed approach in future to detect misplaced fields as well as misplaced methods.

10.5 Detection of Other Code Smells

Although the proposed approach is generic, in this paper we apply it to feature envy and long methods only. The underling rationale, i.e., deep learning techniques could learn useful features for smell detection and the required training data could be generated automatically, may be applicable to other code smells as well. In future it would be interesting to apply the proposed approach to detect more categories of code smells, like *data clumps* and *lazy class*.

11 Conclusions and Future Work

In this paper we propose a deep learning based approach to detect code smells. The key insight of the approach is that advanced deep learning techniques are capable of selecting useful feature for code smell detection, and mapping such features to binary prediction (smelly or non-smelly). To reach the maximal potential of deep learning based code smell detection, we propose an automatic approach to generating labeled training data for smell detection. Compared to traditional datasets created manually, the generated dataset is much larger, which makes it practical to detect code smells with deep learning techniques. As an initial application of the proposed approach, we apply it to four categories of code smells, i.e., feature envy, long method, large class, and misplaced class. The evaluation is composed of two parts. In the first part, we evaluate it on open-source applications with injected smells. In the second part, we evaluate it on the original source code of open-source applications without any revision. Evaluation results in both parts suggest that the proposed approach significantly improves the state-of-the-art in detection of feature envy, long methods, large classes, and misplaced classes.

In future, it should be interesting to apply the proposed approach to detect additional categories of code smells. It is also valuable in future to integrate the implementation of the proposed approach into mainstream IDEs (especially open-source ones). On one side, the integration may benefit developers who are looking for refactoring opportunities. On the other side, it may reveal whether the proposed approach is useful in the real industry.

Acknowledgment

The authors would like to say thanks to the anonymous reviewers for their valuable suggestions. Constructive comments and suggestions from ASE'18 reviewers and PC chairs on how to revise and extend the original version are highly appreciated as well.

The work is partially supported by the National Natural Science Foundation of China (61772071, 61690205).

References

[1] T. Mens and T. Tourwé, "A survey of software refactoring," *IEEE Transactions on Software Engineering*, vol. 30, no. 2, pp. 126–139, 2004.

[2] W. F. Opdyke, "Refactoring object-oriented frameworks," Ph.D. dissertation, University of Illinois at Urbana-Champaign, 1992.

[3] W. G. Griswold and D. Notkin, "Automated assistance for program restructuring," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 2, no. 3, pp. 228–269, July 1993.

[4] F. Tip, A. Kiezun, and D. Baeumer, "Refactoring for generalization using type constraints," in *Proceedings of the Eighteenth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'03)*, Anaheim, CA, October 2003, pp. 13–26.

[5] T. Mens, N. V. Eetvelde, and S. Demeyer, "Formalizing refactorings with graph transformations," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 17, no. 4, pp. 247–276, 2005.

[6] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*. Addison Wesley Professional, 1999.

[7] D. Silva, N. Tsantalis, and M. T. Valente, "Why we refactor? confessions of GitHub contributors," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016. New York, NY, USA: ACM, 2016, pp. 858–870. [Online]. Available: http://doi.acm.org/10.1145/2950290.2950305

[8] H. Liu, X. Guo, and W. Shao, "Monitor-based instant software refactoring," *IEEE Transactions on Software Engineering*, vol. 39, no. 8, pp. 1112–1126, 2013.

[9] F. Palomba, A. Panichella, A. D. Lucia, R. Oliveto, and A. Zaidman, "A textual-based technique for smell detection," in *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, May 2016, pp. 1–10.

[10] M. Zhang, T. Hall, and N. Baddoo, "Code bad smells: a review of current knowledge," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 23, no. 3, pp. 179–202, 2011. [Online]. Available: http://dx.doi.org/10.1002/smr.521

[11] J. A. Dallal, "Identifying refactoring opportunities in object-oriented code: A systematic literature review," *Information and Software Technology*, vol. 58, pp. 231 – 249, 2015. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0950584914001918

[12] T. Sharma and D. Spinellis, "A survey on software smells," *Journal of Systems and Software*, vol. 138, pp. 158 – 173, 2018.

[13] H. Liu, Q. Liu, Z. Niu, and Y. Liu, "Dynamic and automatic feedback-based threshold adaptation for code smell detection," *IEEE Transactions on Software Engineering*, vol. 42, no. 6, pp. 544–558, June 2016.

[14] 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 *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, March 2018, pp. 612–621.

[15] M. V. Mäntylä and C. Lassenius, "Subjective evaluation of software evolvability using code smells: An empirical study," *Empirical Software Engineering*, vol. 11, no. 3, pp. 395–431, Sep 2006. [Online]. Available: https://doi.org/10.1007/s10664-006-9002-8

[16] F. Arcelli Fontana, M. V. Mäntylä, M. Zanoni, and A. Marino, "Comparing and experimenting machine learning techniques for code smell detection," *Empirical Software Engineering*, vol. 21, no. 3, pp. 1143–1191, Jun 2016. [Online]. Available: https://doi.org/10.1007/s10664-015-9378-4

[17] F. A. Fontana and M. Zanoni, "Code smell severity classification using machine learning techniques," *Knowledge-Based Systems*, vol. 128, pp. 43–58, 2017.

[18] 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 *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, March 2018, pp. 612–621.

[19] D. Wu, N. Sharma, and M. Blumenstein, "Recent advances in video-based human action recognition using deep learning: A review," in *2017 International Joint Conference on Neural Networks (IJCNN)*, May 2017, pp. 2865–2872.

[20] H. Palangi, L. Deng, Y. Shen, J. Gao, X. He, J. Chen, X. Song, and R. Ward, "Deep sentence embedding using long short-term memory networks: Analysis and application to information retrieval," *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, vol. 24, no. 4, pp. 694–707, April 2016.

[21] Y. Pan, T. Mei, T. Yao, H. Li, and Y. Rui, "Jointly modeling embedding and translation to bridge video and language," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016, pp. 4594–4602.

[22] K. Hwang and W. Sung, "Character-level incremental speech recognition with recurrent neural networks," in *2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, March 2016, pp. 5335–5339.

[23] X. Gu, H. Zhang, D. Zhang, and S. Kim, "Deep API learning," in *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*. New York, NY, USA: ACM, 2016, pp. 631–642. [Online]. Available: http://doi.acm.org/10.1145/2950290.2950334

[24] H. Liu, Z. Xu, and Y. Zou, "Deep learning based feature envy detection," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE)*. New York, NY, USA: ACM, 2018, pp. 385–396. [Online]. Available: http://doi.acm.org/10.1145/3238147.3238166

[25] S. Wang, L. L. Minku, and X. Yao, "Resampling-based ensemble methods for online class imbalance learning," *IEEE Transactions on Knowledge and Data Engineering*, vol. 27, no. 5, pp. 1356–1368, 2015. [Online]. Available: https://doi.org/10.1109/TKDE.2014.2345380

[26] O. Sagi and L. Rokach, "Ensemble learning: A survey," *Data Mining and Knowledge Discovery*, vol. 8, no. 4, 2018. [Online]. Available: https://doi.org/10.1002/widm.1249

[27] J. Kreimer, "Adaptive detection of design flaws," *Electronic Notes in Theoretical Computer Science*, vol. 141, no. 4, pp. 117–136, 2005.

[28] S. Vaucher, F. Khomh, N. Moha, and Y.-G. Guéhéneuc, "Tracking design smells: Lessons from a study of god classes," in *Reverse Engineering, 2009. WCRE'09. 16th Working Conference on*. IEEE, 2009, pp. 145–154.

[29] F. Khomh, S. Vaucher, Y.-G. Guéhéneuc, and H. Sahraoui, "A bayesian approach for the detection of code and design smells," in *Quality Software, 2009. QSIC'09. 9th International Conference on*. IEEE, 2009, pp. 305–314.

[30] ——, "BDTEX: A GQM-based bayesian approach for the detection of antipatterns," *Journal of Systems and Software*, vol. 84, no. 4, pp. 559–572, 2011.

[31] A. Maiga, N. Ali, N. Bhattacharya, A. Sabané, Y.-G. Guéhéneuc, G. Antoniol, and E. Aïmeur, "Support vector machines for anti-pattern detection," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2012, pp. 278–281.

[32] A. Maiga, N. Ali, N. Bhattacharya, A. Sabane, Y.-G. Gueheneuc, and E. Aimeur, "Smurf: A svm-based incremental anti-pattern detection approach," in *Reverse engineering (WCRE), 2012 19th working conference on*. IEEE, 2012, pp. 466–475.

[33] L. Amorim, E. Costa, N. Antunes, B. Fonseca, and M. Ribeiro, "Experience report: Evaluating the effectiveness of decision trees for detecting code smells," in *Software Reliability Engineering (ISSRE), 2015 IEEE 26th International Symposium on*. IEEE, 2015, pp. 261–269.

[34] F. A. Fontana, M. Zanoni, A. Marino, and M. V. Mantyla, "Code smell detection: Towards a machine learning-based approach," in *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*. IEEE, 2013, pp. 396–399.

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

[36] G. Bavota, R. Oliveto, M. Gethers, D. Poshyvanyk, and A. De Lucia, "Methodbook: Recommending move method refactorings via relational topic models," *Software Engineering, IEEE Transactions on*, vol. 40, no. 7, pp. 671–694, July 2014.

[37] F. Simon, F.Steinbrucker, and C.Lewerentz, "Metrics based refactoring," in *Proceedings of Europen Conference on Software Maintenance and Reengineering*, 2001, pp. 30–38.

[38] O. Seng, J. Stammel, and D. Burkhart, "Search-based determination of refactorings for improving the class structure of object-oriented systems," in *In Proceedings of the 8th annual conference on genetic and evolutionary computation*, 2006, pp. 1909–1916.

[39] N. Tsantalis and A. Chatzigeorgiou, "Identification of move method refactoring opportunities," *IEEE Transactions on Software Engineering*, vol. 35, no. 3, pp. 347–367, 2009.

[40] V. Sales, R. Terra, L. Miranda, and M. Valente, "Recommending move method refactorings using dependency sets," in *Reverse Engineering (WCRE), 2013 20th Working Conference on*, Oct 2013, pp. 232–241.

[41] R. Terra, M. T. Valente, S. Miranda, and V. Sales, "Jmove: A novel heuristic and tool to detect move method refactoring opportunities," *Journal of Systems and Software*, vol. 138, pp. 19 – 36, 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121217302960

[42] F. Palomba, A. Panichella, A. D. Lucia, R. Oliveto, and A. Zaidman, "A textual-based technique for smell detection," in *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, May 2016, pp. 1–10.

[43] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, D. Poshyvanyk, and A. De Lucia, "Mining version histories for detecting code smells," *Software Engineering, IEEE Transactions on*, vol. 41, no. 5, pp. 462–489, May 2015.

[44] H. Liu, Y. Wu, W. Liu, Q. Liu, and C. Li, "Domino effect: Move more methods once a method is moved," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1, March 2016, pp. 1–12.

[45] R. Marinescu, "Detection strategies: Metrics-based rules for detecting design flaws," in *Proceedings of the 20th IEEE International Conference on Software Maintenance*, 2004, pp. 350–359.

[46] M. Lanza and R. Marinescu, *Object-Oriented Metrics in Practice*. Springer, 2006.

[47] S. Charalampidou, A. Ampatzoglou, and P. Avgeriou, "Size and cohesion metrics as indicators of the long method bad smell: An empirical study," in *Proceedings of the 11th International Conference on Predictive Models and Data Analytics in Software Engineering*, ser. PROMISE '15. New York, NY, USA: ACM, 2015, pp. 8:1–8:10. [Online]. Available: http://doi.acm.org/10.1145/2810146.2810155

[48] N. Moha, Y.-G. Gueheneuc, L. Duchien, and A.-F. Le Meur, "Decor: A method for the specification and detection of code and design smells," *Software Engineering, IEEE Transactions on*, vol. 36, no. 1, pp. 20 –36, jan.-feb. 2010.

[49] N. Yoshida, M. Kinoshita, and H. Iida, "A cohesion metric approach to dividing source code into functional segments to improve maintainability," in *2012 16th European Conference on Software Maintenance and Reengineering*, March 2012, pp. 365–370.

[50] S. Charalampidou, E. Arvanitou, A. Ampatzoglou, P. Avgeriou, A. Chatzigeorgiou, and I. Stamelos, "Structural quality metrics as indicators of the long method bad smell: An empirical study," in *2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, Aug 2018, pp. 234–238.

[51] N. Tsantalis and A. Chatzigeorgiou, "Identification of extract method refactoring opportunities," in *2009 13th European Conference on Software Maintenance and Reengineering*, March 2009, pp. 119–128.

[52] ——, "Identification of extract method refactoring opportunities for the decomposition of methods," *Journal of Systems and Software*, vol. 84, no. 10, pp. 1757–1782, 2011.

[53] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Trans. Program. Lang. Syst.*, vol. 9, no. 3, pp. 319–349, 1987. [Online]. Available: https://doi.org/10.1145/24039.24041

[54] K. Maruyama, "Automated method-extraction refactoring by using block-based slicing," in *ACM SIGSOFT Software Engineering Notes*, vol. 26, no. 3. ACM, 2001, pp. 31–40.

[55] D. Silva, R. Terra, and M. T. Valente, "Recommending automated extract method refactorings," in *Proceedings of the 22Nd International Conference on Program Comprehension*, ser. ICPC 2014. New York, NY, USA: ACM, 2014, pp. 146–156. [Online]. Available: http://doi.acm.org/10.1145/2597008.2597141

[56] S. Charalampidou, A. Ampatzoglou, A. Chatzigeorgiou, A. Gkortzis, and P. Avgeriou, "Identifying extract method refactoring opportunities based on functional relevance," *IEEE Transactions on Software Engineering*, vol. 43, no. 10, pp. 954–974, 2017.

[57] S. Xu, A. Sivaraman, S. Khoo, and J. Xu, "Gems: An extract method refactoring recommender," in *IEEE 28th International Symposium on Software Reliability Engineering (ISSRE)*, Oct 2017, pp. 24–34.

[58] C. Marinescu, R. Marinescu, P. F. Mihancea, D. Ratiu, and R. Wettel, "iplasma: An integrated platform for quality assessment of object-oriented design." pp. 77–80, 2005.

[59] F. Palomba, A. Panichella, A. De Lucia, R. Oliveto, and A. Zaidman, "A textual-based technique for smell detection," in *24th IEEE International Conference on Program Comprehension (ICPC)*, May 2016, pp. 1–10.

[60] D. Bobkov, S. Chen, R. Jian, M. Z. Iqbal, and E. Steinbach, "Noise-resistant deep learning for object classification in three-dimensional point clouds using a point pair descriptor," *IEEE Robotics and Automation Letters*, vol. 3, no. 2, pp. 865–872, April 2018.

[61] V. Arnaoudova, L. M. Eshkevari, M. D. Penta, R. Oliveto, G. Antoniol, and Y. G. Guéhéneuc, "Repent: Analyzing the nature of identifier renamings," *IEEE Transactions on Software Engineering*, vol. 40, no. 5, pp. 502–532, May 2014.

[62] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Advances in Neural Information Processing Systems 26*, C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2013, pp. 3111–3119.

[63] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *CoRR*, vol. abs/1301.3781, 2013. [Online]. Available: http://arxiv.org/abs/1301.3781

[64] J. Jin, Z. Xu, and Y. Bu. (2019) DeepSmellDetector. [Online]. Available: https://github.com/liuhuigmail/DeepSmellDetection

[65] K. Zhang, W. Zuo, Y. Chen, D. Meng, and L. Zhang, "Beyond a gaussian denoiser: Residual learning of deep cnn for image denoising," *IEEE Transactions on Image Processing*, vol. 26, no. 7, pp. 3142–3155, July 2017.

[66] S. Ren, K. He, R. Girshick, and J. Sun, "Faster r-cnn: Towards real-time object detection with region proposal networks," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 39, no. 6, pp. 1137–1149, June 2017.

[67] Keras. (2018) Flatten Layer. [Online]. Available: https://github.com/keras-team/keras/blob/master/keras/layers/core.py\#L467

[68] ——. (2018) Merge Layer. [Online]. Available: https://github.com/keras-team/keras/blob/master/keras/layers/merge.py

[69] ——. (2018) Keras: The python deep learning library. [Online]. Available: https://github.com/keras-team/keras/blob/master/keras/models.py

[70] M. Lanza and R. Marinescu, *Object-Oriented Metrics in Practice - Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*. Springer, 2006. [Online]. Available: https://doi.org/10.1007/3-540-39538-5

[71] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476 – 493, 1994.

[72] J. Bansiya and C. G. Davis, "A hierarchical model for object-oriented design quality assessment," *IEEE Transactions on Software Engineering*, vol. 28, no. 1, pp. 4–17, January 2002.

[73] *JUnit*, https://junit.org/, 2018.

[74] *PMD*, https://pmd.github.io/, 2018.

[75] *JExcelAPI*, http://jexcelapi.sourceforge.net/, 2018.

[76] *Areca*, http://www.areca-backup.org/, 2018.

[77] *Freeplane*, https://www.freeplane.org/, 2018.

[78] *jEdit*, http://www.jedit.org/, 2018.

[79] *Weka*, http://www.cs.waikato.ac.nz/ml/weka/, 2018.

[80] *Android Backup Extractor*, https://sourceforge.net/projects/adbextract 2019.

[81] *Android Backup Extractor*, http://www.artofillusion.org/, 2019.

[82] *Grinder*, http://grinder.sourceforge.net/, 2019.

[83] W. F. Opdyke, "Refactoring object-oriented frameworks," Ph.D. dissertation, Champaign, IL, USA, 1992, uMI Order No. GAX93-05645.

[84] F. Palomba, G. Bavota, M. D. Penta, F. Fasano, R. Oliveto, and A. D. Lucia, "On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation," *Empirical Software Engineering*, Aug 2017. [Online]. Available: https://doi.org/10.1007/s10664-017-9535-z

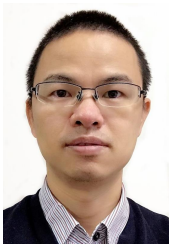
[85] *XDM*, http://xdman.sourceforge.net/, 2018.

[86] *JSmooth*, http://jsmooth.sourceforge.net/, 2018.

[87] *Neuroph*, http://neuroph.sourceforge.net/, 2018.

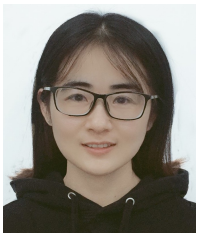
[88] *DavMail*, http://davmail.sourceforge.net/, 2019.

[89] *CKEditor*, https://ckeditor.com/, 2019.



Hui Liu is a professor at the School of Computer Science and Technology, Beijing Institute of Technology, China. He received BS degree in control science from Shandong University in 2001, MS degree in computer science from Shanghai University in 2004, and PhD degree in computer science from the Peking University in 2008. He was a visiting research fellow in centre for research on evolution, search and testing (CREST) at University College London, UK. He served on the program committees and organizing committees of prestigious conferences, such as ICSME, RE, ICSR, and COMPSAC. He is particularly interested in software refactoring, AI-based software engineering, and software quality. He is also interested in developing practical tools to assist software engineers.

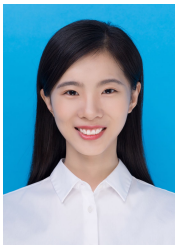
Jiahao Jin received BS degree from Gengdan Institute of Beijing University of Technology in computer science in 2017. He is currently working toward the master's degree at the School of Computer Science and Technology, Beijing Institute of Technology, under the supervision of Dr. Hui Liu. He is particularly interested in software refactoring and software evolution.



Zhifeng Xu received BS degree from Northeast Forestry University in computer science in 2016. She is currently working toward the master's degree at the School of Computer Science and Technology, Beijing Institute of Technology, under the supervision of Dr. Hui Liu. She is interested in software refactoring and software evolution.

Yanzhen Zou is an associate professor in Peking University, Beijing, China. She received received her PhD degree from the Peking University in 2010. She was a visiting research fellow in Department of Computer Science, Toronto University, Canada. Her past projects have investigated a range of problems surrounding software reuse, including component based software development, code search, component quality assessment, software data mining, etc. She is currently working on software knowledge graph, API understanding and auto-comment, API reuse oriented code synthesis and developing practical software tools.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2019.2936376, IEEE Transactions on Software Engineering



Yifan Bu received BS degree from Central South University in Software Engineering in 2016. She is currently working toward the master's degree at the School of Computer Science and Technology, Beijing Institute of Technology, under the supervision of Dr. Hui Liu. She is interested in software refactoring and software evolution.

Lu Zhang is a professor at the School of Electronics Engineering and Computer Science, Peking University, P.R. China. He received both PhD and BSc in Computer Science from Peking University in 2000 and 1995 respectively. He was a postdoctoral researcher in Oxford Brookes University and University of Liverpool, UK. He served on the program committees of many prestigious conferences, such as FSE, OOPSLA, ISSTA and ASE. He was a program co-chair of SCAM2008 and a program co-chair of ICSM17. He has been on the editorial boards of Journal of Software Maintenance and Evolution: Research and Practice and Software Testing, Verification and Reliability. His current research interests include software testing and analysis, program comprehension, software maintenance and evolution, software reuse, and program synthesis.