Deep Learning Based Feature Envy Detection

Hui Liu*
School of Computer Science and Technology, Beijing Institute of Technology
Beijing, China
Liuhui08@bit.edu.cn

Zhifeng Xu
School of Computer Science and Technology, Beijing Institute of Technology
Beijing, China
848602422@qq.com

Yanzhen Zou†
Key Laboratory of High Confidence Software Technologies (Peking University),Ministry of Education
Beijing, China
zouyz@pku.edu.cn

ABSTRACT

Software refactoring is widely employed to improve software quality. A key step in software refactoring is to identify which part of the software should be refactored. To facilitate the identification, a number of approaches have been proposed to identify certain structures in the code (called code smells) that suggest the possibility of refactoring. Most of such approaches rely on manually designed heuristics to map manually selected source code metrics to predictions. However, it is challenging to manually select the best features, especially textual 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 feature envy, one of the most common code smells. The key insight is that deep neural networks and advanced deep learning techniques could automatically select features (especially textual features) of source code for feature envy detection, and could automatically build the complex mapping between such features and predictions. We also propose an automatic approach to generating labeled training data for the neural network based classifier, which does not require any human intervention. Evaluation results on open-source applications suggest that the proposed approach significantly improves the state-of-the-art in both detecting feature envy smells and recommending destinations for identified smelly methods.

CCS CONCEPTS

• **Software and its engineering → Software maintenance tools**; **Object oriented development**;

KEYWORDS

Feature Envy, Deep Learning, Software Refactoring, Code Smells

ACM Reference Format:
Hui Liu, Zhifeng Xu, and Yanzhen Zou. 2018. Deep Learning Based Feature Envy Detection. In *Proceedings of the 2018 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18), September 3–7, 2018, Montpellier, France.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3238147.3238166

*corresponding author
†Also with School of Electronics Engineering and Computer Science, Peking University.

1 INTRODUCTION

Software refactoring is widely employed to improve software quality by restructuring its internal structures whereas its external behaviors are kept unchanged [37, 42]. Most of the modern IDEs provide tool support for software refactoring. 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 [20, 36, 56].

A key step in software refactoring is to identify where refactorings should be applied [37]. To facilitate the identification, Beck and Fowler [19] 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 *large class*. They also analyze their features, as well as their impact and potential solutions (refactorings).

However, it is tedious and time consuming to manually identify code smells, especially when such code smells involve more than one file or package [29, 37]. Consequently, a large number of automatic or semi-automatic approaches have been proposed to detect different kinds of code smells [37, 47, 63]. Detailed literature review and analysis on code smell detection have been made by Zhang et al. [63], Dallal [15], and more recently by Sharma and Spinellis [53].

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 [15, 30, 63]. However, it is challenging to manually select the best features, especially textual features. It is also difficult to manually construct the optimal heuristics. Analysis results on such approaches [15, 40, 63] 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 [35]. 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 [11, 17]. However, empirical studies [41] 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 *feature envy*, one of the most common code smells. The key insight is that deep neural networks and advanced deep learning techniques could automatically select useful features from source code (especially textual features) 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 to be good at selecting useful features and building complex mapping from input to output automatically [60]. With significant advances in deep learning techniques, they have been successfully used in different domains, e.g., natural language processing (NLP) [44], video processing [49], speech recognition [22], and software engineering [21]. That is the reason why we employ deep learning techniques in this paper to build a neural network based classifier that classifies methods in subject applications into '*smelly*' and '*non-smelly*'.

To train the neural network based classifier, we also propose an automatic approach to generating labeled training data without any human intervention. One of the biggest challenges for machine learning (especially deep learning) based smell detectors is to collect a large number of labeled samples to train the classifiers (detectors). To collect labeled training data, existing approaches often rely heavily on manual checking of the initial detection results of similar smell detectors [11, 40]. However, manual checking is time consuming, which significantly limits the size of labeled training data, and thus prevents machine learning techniques from reaching their maximal potential. To this end, in this paper we generate labeled samples automatically based on open-source applications. We generate negative samples by extracting methods (and their context) directly from high quality open-source applications, assuming that such methods are correctly placed in the original programs. To generate positive samples, we move methods randomly to other classes where it could be moved by refactoring tools. After the movement, such methods (together with their new context) are taken as smelly methods (with *feature envy*). We can generate a large number of such labeled training data because the generation is fully automatic. The large data in turn serve as the basis of machine learning based smell detection.

The evaluation of the proposed approach is composed of two parts. In the first part, we evaluate it on 7 well-known open-source applications with automatically injected feature envy smells. Evaluation results suggest that the proposed approach significantly outperforms existing approaches. It improves F-measure by 34.32%. In the second part, we evaluate it on 3 open-source applications where no smells are injected. Evaluation results also suggest that the proposed approach significantly improves the state-of-the-art.

The paper makes the following contributions:

- A deep learning based approach to identifying feature envy. To the best of our knowledge, we are the first to apply deep learning techniques to feature envy detection.
- An automatic approach to generating labeled training data for feature envy detection.
- Evaluation of the proposed approach whose results suggest that the proposed approach can significantly improve the state-of-the-art.

The rest of the paper is structured as follows. Section 2 introduces related research. Section 3 proposes the approach. Section 4 and Section 5 present the evaluation of the proposed approach. Section 6 discusses related issues. Section 7 makes conclusions.

2 RELATED WORK

2.1 Feature Envy

Beck and Fowler [19] 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 [13]. The first one was proposed by Simon et al. in 2001 [54]. They define a distance to measure how closely two entities are related:

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

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. [52] propose a search based approach to identify move method opportunities. They define a fitness function:

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

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 [57] 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:

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

Otherwise, the distance is computed as follows:

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

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. [54]. 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.[51, 55] 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 this 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. [13] 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. It is the first approach to identify move method opportunities based on textual information besides source code structures. Palomba et al. [48] 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. [45] 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. [31] 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.

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.

2.2 Machine Learning Based Smell Detection

With the advance in machine learning techniques, a number of machine learning based smell detection approaches are proposed [11]. Kreimer [28] proposes a decision tree based approach to identify code smells, e.g, *long method* and *large class*. Vaucher et al. [26, 27, 58] apply Bayesian beliefs networks to detect *God class* (*Blob class*). Maiga et al. [33, 34] exploit Support Vector Machine (SVM) in detection of *Blob class*, and the same technology is employed by Amorim et al. [10] to detect *Blob class*, *long parameter list*, *long method*, and *feature envy*. Fontana et al. [16–18] 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 [41]. 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.

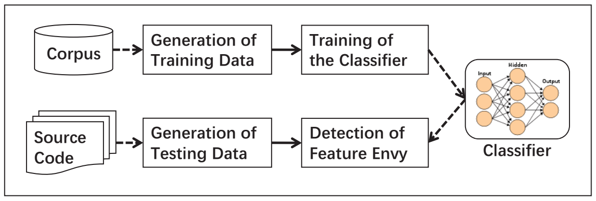
Corpus
Generation of Training Data
Training of the Classifier
Source Code
Generation of Testing Data
Detection of Feature Envy
Input
Hidden
Output
Classifier

Figure 1: Overview of the Proposed Approach

3 APPROACH

In this section, we propose a deep learning based approach to identify feature envy. 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 training samples (i.e., methods with or without feature envy). Such training samples are employed to train a neural network based classifier whose output indicates whether the input method m from class ec envies another class tc. At prediction phase, for a given method m we select a set of potential target classes $tc = \{c_1, c_2 \ldots, c_k\}$, and predict with the resulting neural network whether m should be moved to any of them. Details of the proposed approach are presented in the following sections.

3.2 Input

To decide wether 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 [57]. We reuse such metrics because of the following reasons:

- First, they have been proved effective in feature envy detection [57].
- Second, the open-source implementation of *JDeodorant* makes it easy to extract such metrics.

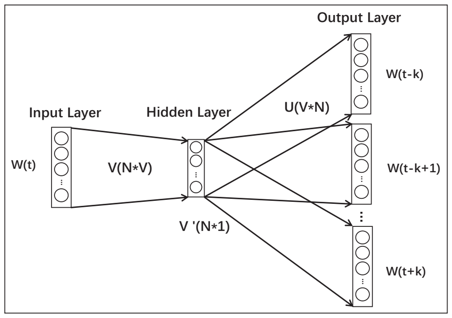

Figure 2: Model of Word2Vector

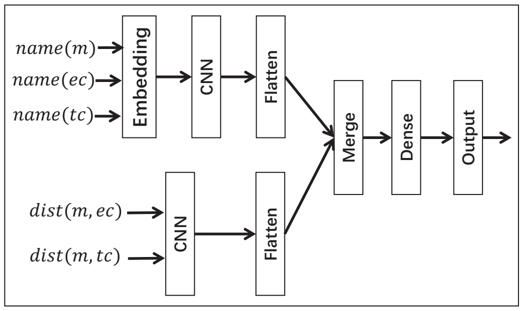

Figure 3: Neural Network based Classifier

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 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 [12].

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

$$\begin{aligned} input \;\; = \;\; & < name(m), name(ec), name(tc), \\ & dist(m, ec), dist(m, tc) > \end{aligned} \tag{5}$$

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. 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.

3.3 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. [38, 39]. *Word2vector* has been proved efficient for learning high-quality distributed vector representations that capture precise syntactic and semantic word relationships [39]. *Word2vector* is essentially a neural network that predicts nearby words, i.e., words before and after it (as shown in Fig 2). 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:

$$\begin{aligned} name(e) \;\; & = \;\; < w_1, w_2, \ldots, w_k > && (6) \\ & = \;\; < V(w_1), V(w_2), \ldots, V(w_k) > && (7) \end{aligned}$$

$name(e)$ 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.

3.4 Deep Neural Network based Classifier

The structure of the deep neural network based classifier is presented in Fig. 3 and source code is available online [61]. 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 3.3. Such numerical vectors are in turn fed into a Convolutional Neural Network (CNN). In total we have tree 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 [62]. 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 [50]. The output of CNN is forwarded to a *flatten* layer [23] 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. Notable, 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 (for space limitation, details are presented at [32]). The textual input and the numerical input are finally merged by the *merge* layer [25] 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.

3.5 Generation of Training Data

Deep neural networks have a large number of parameters. Consequently, they often require a large number of training data to adjust such parameters. To train the deep neural network proposed in Section 3.4, we generate training data as follows. First, we download well-known and high quality open-source applications. Second, for each method m 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. (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:

$$\begin{aligned} ngItem &= < input, output > && (8)\\ input &= < name(m), name(ec), name(tc_i), \\ &\quad dist(m, ec), dist(m, tc_i) > && (9)\\ output &= 0 && (10)\end{aligned}$$

(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.

The generation is based on the assumption that all involved methods are correctly placed in the original applications. Consequently, the original methods if not moved should be taken as non-smelly (without feature envy). In contrast, if they are moved in some way by move method refactorings, they should be taken as smelly (with feature envy) after the movement. However, the assumption may not hold in some cases, i.e., some methods in the subject applications may have been placed improperly. As a result, the training data generated based on such methods 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 methods are placed correctly. As a result, most of the training items could be labeled correctly. Second, advanced neural networks like CNN work well even if the training data contain a few mislabeled items [14].

3.6 Feature Envy Detection

3.6.1 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 5) 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).

3.6.2 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 3.4 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 [24]. 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.

4 EVALUATION

In this section, we evaluate the proposed approach on seven open-source applications with injected feature envy smells.

4.1 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**: 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?

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 *JDeodorant* [57] and *JMove* [55]. They are selected for comparison because of the following reasons. First, they represent the state-of-the-art. *JDeodorant* has been widely employed as a benchmark [13, 31, 55] whereas *JMove* is one of the latest advances in this area. Second, they are 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 smelly 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 *Q3* 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.

4.2 Subject Applications

We evaluate the proposed approach on seven 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, version, num of classes (NOC), number of methods (NOM), and lines of source code (LOC), respectively.

JUnit [6] is a widely used testing framework. During the last 16 years, it has released more than 40 versions. *PMD* [8] 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* [4] 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* [1] 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* [2] is a free mind mapping and knowledge management software. During the last 18 years, it has released more than 100 versions. *jEdit* [3] is a free text editor. During the last 18 years, it has released 143 versions. *Weka* [59] 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.

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, popular and of high quality. All of them have involved successfully for a long time (more than 7 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. As introduced in Section 3.5, the generation of training data is based on the assumption that methods in subject applications are correctly placed. Consequently, selecting such high quality applications may reduce the likelihood that methods in such applications are misplaced. Notable, the average LCM (Lack of Cohesion of Methods) of such projects is smaller than 0.26, the average efferent coupling is smaller than 11, and the average afferent coupling is smaller than 30. All of these together may suggest that most methods have benn placed correctly. Third, these applications were developed by different developers, which may reduce the bias introduced by specific developers.

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

4.3 Process

We carry out a k-fold ($k = 7$) cross-validation on the seven subject applications presented in Table 1. On each fold, a single application is used as the testing subject (noted as *testingApp*) whereas the others are used as training subjects (noted as *trainingApps*). Each of the subject applications is used as the testing subject for once. 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 3.5.
(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 *JDK' 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, *JDeodorant*, and *JMove* 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 [43]. 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. [46] 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% × 23% of all methods (including unmovable ones), which is consistent with the finding reported by Palomba et al. [46].

After the evaluation, we calculate the precision, recall, and F_1 of the proposed approach (*JDeodorant* and *JMove* as well) in identifying feature envy smells as follows:

$$precision = \frac{true\ positives}{true\ positives\ +\ false\ negatives} \tag{11}$$

$$recall = true\ positives \div |ms_1| \tag{12}$$

$$F_1 = 2 \times \frac{precision \times recall}{precision + recall} \tag{13}$$

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

$$ac = \frac{correct\ recommendation\ for\ true\ positives}{true\ positives} \tag{14}$$

The recommended destination for method m is correct if and only if it suggests to move m back to its enclosing class 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 should be moved (i.e., it is a true positive) they should consider the destination problem.

4.4 RQ1:Detection of Feature Envy

To answer research question **RQ1** we compare the proposed approach against *JDeodorant* and *JMove* in detecting feature envy smells. Evaluation results are presented in Table 2. The first column presents the subject applications. Columns 2-4 presents the precision, recall, and F-measure of the proposed approach, respectively. The performance of *JDeodorant* and *JMove* is presented in columns 5-7 and columns 8-10, respectively. 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 F-measure is concerned. Its average F-measure is 52.98% whereas the average F-measure of *JDeodorant* and *JMove* is 18.66% and 17.27%, respectively. Compared to *JDeodorant* and *JMove*, the proposed approach improves F-measure by 34.32% (=52.98%-18.66%) and 35.71% (=52.98%-17.27%), respectively.
- Second, the proposed approach can identify most of the feature envy smells. Its average recall is up to 79.27%. Compared to *JDeodorant* and *JMove*, it improves recall dramatically by 67.05% (=79.27%-12.22%) and 62.97% (=79.27%-16.3%), respectively.
- Third, the precision (39.79%) of the proposed approach is slightly greater than that (39.51%) of *JDeodorant*, and it is significantly greater than that (18.37%) of *JMove*.

We evaluate how the textual input and code metrics contribute to the proposed model by removing them respectively. Evaluation results suggest that removing the textual input significantly reduces recall of the approach from 79.27% to 46.69%, and removing the code metrics reduces precision significantly from 39.76% to 23.20%.

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.

4.5 RQ2:Recommendation of Destinations

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 69.09% to 80.52%. On average, its accuracy is up to 74.94%. In other words, for 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* and *JMove* in recommending destinations for smelly methods. Compared to *JDeodorant* and *JMove*, it improves the accuracy by 27.15% (=74.94%-47.79%) and 46.54% (=74.94%-28.4%), respectively.

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.

4.6 RQ3: 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

Table 2: Evaluation Results on Feature Envy Detection

Applications	Proposed Approach			JDeodorant			JMove		
	precision	*recall*	F_1	*precision*	*recall*	F_1	*precision*	*recall*	F_1
JUnit	40.59%	91.11%	56.16%	30.76%	14.82%	20%	22.72%	18.52%	20.41%
PMD	41.27%	68.42%	51.49%	15.79%	5.36%	8%	30%	26.79%	28.3%
JExcelAPI	31.9%	92.85%	47.49%	60%	10.7%	18.18%	27.27%	16.07%	20.22%
Areca	46.05%	72.16%	56.23%	32.14%	9.28%	14.4%	26.76%	39.18%	31.8%
Freeplane	38.09%	68.58%	48.97%	21.62%	8.94%	12.65%	24.83%	13.79%	17.73%
jEdit	42.63%	78.57%	55.28%	22.73%	4.55%	7.58%	17.43%	13.57%	15.26%
Weka	40.05%	86%	54.65%	58.33%	17.5%	26.92%	11.22%	11.75%	11.48%
Average	**39.79%**	**79.27%**	**52.98%**	**39.51%**	**12.22%**	**18.66%**	**18.37%**	**16.3%**	**17.27%**

Table 3: Accuracy in Recommending Destinations

Applications	Proposed Approach	JDeodorant	JMove
JUnit	70.73%	75%	60%
PMD	76.92%	33.33%	46.67%
JExcelAPI	61.54%	50%	44.44%
Areca	71.43%	55.55%	23.68%
Freeplane	73.74%	31.25%	27.78%
jEdit	69.09%	60%	26.31%
Weka	80.52%	48.57%	21.27%
Average	**74.94%**	**47.79%**	**28.4%**

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 10 minutes. With the trained classifier, it takes 1.7-25.3 minutes (9.43 minutes on average) to detect feature envy smells for one subject application. In contrast, on average it takes *Jdeodorant* and *JMove* 1 minute and 41 minutes respectively. Further analysis on the testing time suggests that most (99%=9.35 minutes/9.43 minutes) of the testing time is spent on information extraction, i.e., extracting method names, class names, and the distance between them. In contrast, the neural network always makes predictions instantly, and it consumes less than 1% of the testing time (less than 5 seconds).

4.7 Threats to Validity

The first threat to validity is that the feature envy smells 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, we randomly select methods to move, and randomly select the target classes for such methods. Besides that, we also rigidly control the ratio of smelly methods (to all methods) to make sure that the ratio is close to that (2.3%) in real applications [46]. Finally, we carry out a case study in Section 5 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 all methods in the subject applications are properly distributed, i.e., there is no feature envy in the original source code. 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 seven 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 k-fold cross validity on such applications to reduce the bias introduced by specific applications. To facilitate further evaluation, we publish the source code of the implementation as well as the evaluation data at https://github.com/liuhuigmail/FeatureEnvy.

5 CASE STUDY

In Section 4, we evaluate the proposed approach on applications with automatically injected smells. In this section, we evaluate it with real applications without any injection. This case study investigates the same research questions as introduced in Section 4.1, but with real applications that have not be changed by us anyway.

5.1 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 ten thousand or greater than thirty 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.

Table 4: 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

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 (as well *JDeodorant* and *JMove*) to extract necessary information correctly. Finally, we select the top three of the remaining subject applications. Table 4 presents the information of such applications.

XDM [9] (Xtreme Download Manager) is a download manager to save streaming videos from websites, resume broken/dead downloads, and schedule downloads. *JSmooth* [5] is a Java executable wrapper that creates native Windows launchers for given Java applications. *Neuroph* [7] is a lightweight neural network framework providing Java neural network library and GUI tool to facilitate creating, training and saving of neural networks.

5.2 Process

For each of the subject applications, we carry out the case study as follows. First, we train the proposed approach with data generated from the 7 subject applications that are introduced in Table 1. Second, we apply the proposed approach (trained classifier), *JDeodorant*, and *JMove* 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 postgraduate 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. Notable, the evaluation is conducted in an anonymous manner, and thus the students cannot identify the tool that performed the recommendation.

5.3 Results

Results of the case study are presented in Table 5. The first column presents different metrics, like precision and accuracy. Columns 2-4 presents the results of the proposed approach on different applications, and the fifth column presents its average performance on all applications. The performance of *JDeodorant* and *JMove* is presented in columns 6-9 and columns 10-13, respectively. 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 5, 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 114 feature envy smells from the three subject applications, among which 47 are manually confirmed. It also successfully recommends the correct destinations for 78.72% (=37/47) of the confirmed smelly methods.
- Second, the proposed approach significantly outperforms *JDeodorant* and *JMove* in detecting feature envy smells. It improves the precision from 27.59% (*JDeodorant*) and 15.35% (*JMove*) significantly to 41.23%. It also identify much more true positives than *JDeodorant* and *JMove*, which suggests that the proposed approach achieves greater recall than *JDeodorant* and *JMove*.
- Third, the proposed approach is more accurate than *JDeodorant* and *JMove* in recommending destinations for feature envy methods. Compared to *JDeodorant* and *JMove*, it improves the accuracy by 41.22% (=78.72%-37.5%) and 48.42% (=78.72%-30.3%), respectively.

We conclude from the preceding analysis that the proposed approach significantly outperform the state-of-the-art in both detecting feature envy smells and recommending destinations for smelly methods.

5.4 Threats to Validity

The first threat to validity is that only three 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. 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 feature envy 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 4 where manual checking is not required.

6 DISCUSSION

6.1 Evaluation with Original Developers

The case study in Section 5 recruits external developers to manually check the potential code smells reported by smell detection tools.

Table 5: Results of Case Study

Metrics	Proposed Approach				JDeodorant				JMove			
	XMD	JSmooth	Neuroph	**Total**	XMD	JSmooth	Neuroph	**Total**	XMD	JSmooth	Neuroph	**Total**
#Reported	32	26	56	**114**	8	3	18	**29**	106	27	82	**215**
#Accepted	15	11	21	**47**	3	1	4	**8**	12	5	16	**33**
#Accepted targets	12	9	16	**37**	1	1	1	**3**	3	2	5	**10**
Precision	46.88%	42.31%	37.5%	**41.23%**	37.5%	33.33%	22.22%	**27.59%**	11.32%	18.52%	19.51%	**15.35%**
Accuracy (destination)	80%	81.82%	76.19%	**78.72%**	33.33%	100%	25%	**37.5%**	25%	40%	31.25%	**30.3%**

Such external developers are not familiar with the subjection 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. [30] 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.

6.2 Generality of the Generated Data

The proposed approach generates labeled training data automatically by randomly moving methods among classes. Such generated data serve as the basis of deep learning based feature envy detection. However, they may be different from real feature envy smells introduced by developers. As a result, the classifier trained with such generated data may learn to distinguish the randomly moved methods instead of those associated with feature envy smells. To reduce the threat, we evaluate the proposed approach with a case study in Section 5 where the detected feature envy smells are manually confirmed. The evaluation results suggest that the proposed approach can identify real feature envy 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.

6.3 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 [19] 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.

6.4 Extension to Other Code Smells

The proposed approach is currently confined to feature envy. However, 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. For example, we may generate God classes (positive samples) by combining classes in the same project, and take other classes as negative samples. With such positive and negative samples (training data), we can train a deep neural network to determine whether a given class is a God class.

7 CONCLUSIONS

In this paper we propose a deep learning based approach to identify feature envy smells, one of the most common code smells. It automatically selects useful textual features, and maps such features to predictions automatically. To train the deep neural network, we also propose an approach to generate numerous labeled training data automatically. To the best of our knowledge, we are the first to exploit deep learning in feature envy detection. We are also the first to automatically generate labeled training data for code smell detection. The evaluation of the proposed approach is composed of two parts. In the fist 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 both feature envy detection and recommendation of refactoring solutions.

ACKNOWLEDGMENTS

The authors would like to say thanks to the reviewers and PC chairs for their insightful comments and constructive suggestions.

This work is supported by the National Natural Science Foundation of China under Grant No.: 61690205, 61472034,61772071, and National Key Research and Development Program of China under Grant No.: 2016YFB1000801

REFERENCES

[1] 2018. *Areca*. http://www.areca-backup.org/.
[2] 2018. *Freeplane*. https://www.freeplane.org/.
[3] 2018. *jEdit*. http://www.jedit.org/.
[4] 2018. *JExcelAPI*. http://jexcelapi.sourceforge.net/.
[5] 2018. *JSmooth*. http://jsmooth.sourceforge.net/.
[6] 2018. *JUnit*. https://junit.org/.
[7] 2018. *Neuroph*. http://neuroph.sourceforge.net/.
[8] 2018. *PMD*. https://pmd.github.io/.
[9] 2018. *XDM*. http://xdman.sourceforge.net/.
[10] Lucas Amorim, Evandro Costa, Nuno Antunes, Baldoino Fonseca, and Márcio Ribeiro. 2015. Experience report: Evaluating the effectiveness of decision trees for detecting code smells. In *Software Reliability Engineering (ISSRE), 2015 IEEE 26th International Symposium on*. IEEE, 261–269.
[11] Francesca Arcelli Fontana, Mika V. Mäntylä, Marco Zanoni, and Alessandro Marino. 2016. Comparing and experimenting machine learning techniques for code smell detection. *Empirical Software Engineering* 21, 3 (01 Jun 2016), 1143–1191. https://doi.org/10.1007/s10664-015-9378-4
[12] V. Arnaoudova, L. M. Eshkevari, M. D. Penta, R. Oliveto, G. Antoniol, and Y. G. GuÃľhÃľneuc. 2014. REPENT: Analyzing the Nature of Identifier Renamings. *IEEE Transactions on Software Engineering* 40, 5 (May 2014), 502–532. https://doi.org/10.1109/TSE.2014.2312942
[13] G. Bavota, R. Oliveto, M. Gethers, D. Poshyvanyk, and A. De Lucia. 2014. Methodbook: Recommending Move Method Refactorings via Relational Topic Models. *Software Engineering, IEEE Transactions on* 40, 7 (July 2014), 671–694. https://doi.org/10.1109/TSE.2013.60
[14] D. Bobkov, S. Chen, R. Jian, M. Z. Iqbal, and E. Steinbach. 2018. Noise-Resistant Deep Learning for Object Classification in Three-Dimensional Point Clouds Using a Point Pair Descriptor. *IEEE Robotics and Automation Letters* 3, 2 (April 2018), 865–872. https://doi.org/10.1109/LRA.2018.2792681
[15] Jehad Al Dallal. 2014. Identifying Refactoring Opportunities in Object-Oriented Code: A Systematic Literature Review. *Information and Software Technology* 0 (2014), –. https://doi.org/10.1016/j.infsof.2014.08.002
[16] Francesca Arcelli Fontana, Mika V Mäntylä, Marco Zanoni, and Alessandro Marino. 2016. Comparing and experimenting machine learning techniques for code smell detection. *Empirical Software Engineering* 21, 3 (2016), 1143–1191.
[17] Francesca Arcelli Fontana and Marco Zanoni. 2017. Code smell severity classification using machine learning techniques. *Knowledge-Based Systems* 128 (2017), 43–58.
[18] Francesca Arcelli Fontana, Marco Zanoni, Alessandro Marino, and Mika V Mantyla. 2013. Code smell detection: Towards a machine learning-based approach. In *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*. IEEE, 396–399.
[19] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. 1999. *Refactoring: Improving the Design of Existing Code*. Addison Wesley Professional.
[20] William G. Griswold and David Notkin. 1993. Automated assistance for program restructuring. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 2, 3 (July 1993), 228–269.
[21] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. 2016. Deep API Learning. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016)*. ACM, New York, NY, USA, 631–642. https://doi.org/10.1145/2950290.2950334
[22] K. Hwang and W. Sung. 2016. Character-level incremental speech recognition with recurrent neural networks. In *2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 5335–5339. https://doi.org/10.1109/ICASSP.2016.7472696
[23] Keras. 2018. Flatten Layer. Retrieved July 21, 2018 from https://github.com/keras-team/keras/blob/master/keras/layers/core.py#L467
[24] Keras. 2018. Keras: The Python Deep Learning Library. https://github.com/keras-team/keras/blob/master/keras/models.py
[25] Keras. 2018. Merge Layer. Retrieved July 21, 2018 from https://github.com/keras-team/keras/blob/master/keras/layers/merge.py
[26] Foutse Khomh, Stéphane Vaucher, Yann-Gaël Guéhéneuc, and Houari Sahraoui. 2009. A bayesian approach for the detection of code and design smells. In *Quality Software, 2009. QSIC'09. 9th International Conference on*. IEEE, 305–314.
[27] Foutse Khomh, Stephane Vaucher, Yann-Gaël Guéhéneuc, and Houari Sahraoui. 2011. BDTEX: A GQM-based Bayesian approach for the detection of antipatterns. *Journal of Systems and Software* 84, 4 (2011), 559–572.
[28] Jochen Kreimer. 2005. Adaptive detection of design flaws. *Electronic Notes in Theoretical Computer Science* 141, 4 (2005), 117–136.
[29] Hui Liu, Xue Guo, and Weizhong Shao. 2013. Monitor-Based Instant Software Refactoring. *IEEE Transactions on Software Engineering* 39, 8 (2013), 1112–1126. https://doi.org/10.1109/TSE.2013.4
[30] H. Liu, Q. Liu, Z. Niu, and Y. Liu. 2016. Dynamic and Automatic Feedback-Based Threshold Adaptation for Code Smell Detection. *IEEE Transactions on Software Engineering* 42, 6 (June 2016), 544–558. https://doi.org/10.1109/TSE.2015.2503740
[31] H. Liu, Y. Wu, W. Liu, Q. Liu, and C. Li. 2016. 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. 1–12. https://doi.org/10.1109/SANER.2016.14
[32] Hui Liu, Zhifeng Xu, and Yanzhen Zou. 2018. Replace CNN with Dense Layers. Retrieved July 21, 2018 from https://github.com/liuhuigmail/FeatureEnvy/tree/master/Algorithm/DenseVScnn
[33] Abdou Maiga, Nasir Ali, Neelesh Bhattacharya, Aminata Sabane, Yann-Gael Gueheneuc, and Esma Aimeur. 2012. SMURF: A SVM-based incremental anti-pattern detection approach. In *Reverse engineering (WCRE), 2012 19th working conference on*. IEEE, 466–475.
[34] Abdou Maiga, Nasir Ali, Neelesh Bhattacharya, Aminata Sabané, Yann-Gaël Guéhéneuc, Giuliano Antoniol, and Esma Aïmeur. 2012. Support vector machines for anti-pattern detection. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. ACM, 278–281.
[35] Mika V. Mäntylä and Casper Lassenius. 2006. Subjective evaluation of software evolvability using code smells: An empirical study. *Empirical Software Engineering* 11, 3 (01 Sep 2006), 395–431. https://doi.org/10.1007/s10664-006-9002-8
[36] Tom Mens, Niels Van Eetvelde, and Serge Demeyer. 2005. Formalizing Refactorings with Graph Transformations. *Journal of Software Maintenance and Evolution: Research and Practice* 17, 4 (2005), 247–276.
[37] Tom Mens and Tom Tourwé. 2004. A Survey of Software Refactoring. *IEEE Transactions on Software Engineering* 30, 2 (2004), 126–139.
[38] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient Estimation of Word Representations in Vector Space. *CoRR* abs/1301.3781 (2013). arXiv:1301.3781 http://arxiv.org/abs/1301.3781
[39] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. 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., 3111–3119.
[40] D. Di Nucci, F. Palomba, D. A. Tamburri, A. Serebrenik, and A. De Lucia. 2018. Detecting code smells using machine learning techniques: Are we there yet?. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 612–621. https://doi.org/10.1109/SANER.2018.8330266
[41] D. Di Nucci, F. Palomba, D. A. Tamburri, A. Serebrenik, and A. De Lucia. 2018. Detecting code smells using machine learning techniques: Are we there yet?. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 612–621. https://doi.org/10.1109/SANER.2018.8330266
[42] William F. Opdyke. 1992. *Refactoring Object-Oriented Frameworks*. Ph.D. Dissertation. University of Illinois at Urbana-Champaign.
[43] William F. Opdyke. 1992. *Refactoring Object-oriented Frameworks*. Ph.D. Dissertation. Champaign, IL, USA. UMI Order No. GAX93-05645.
[44] H. Palangi, L. Deng, Y. Shen, J. Gao, X. He, J. Chen, X. Song, and R. Ward. 2016. Deep Sentence Embedding Using Long Short-Term Memory Networks: Analysis and Application to Information Retrieval. *IEEE/ACM Transactions on Audio, Speech, and Language Processing* 24, 4 (April 2016), 694–707. https://doi.org/10.1109/TASLP.2016.2520371
[45] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, D. Poshyvanyk, and A. De Lucia. 2015. Mining Version Histories for Detecting Code Smells. *Software Engineering, IEEE Transactions on* 41, 5 (May 2015), 462–489. https://doi.org/10.1109/TSE.2014.2372760
[46] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Fausto Fasano, Rocco Oliveto, and Andrea De Lucia. 2017. On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. *Empirical Software Engineering* (07 Aug 2017). https://doi.org/10.1007/s10664-017-9535-z
[47] F. Palomba, A. Panichella, A. De Lucia, R. Oliveto, and A. Zaidman. 2016. A textual-based technique for Smell Detection. In *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*. 1–10. https://doi.org/10.1109/ICPC.2016.7503704
[48] F. Palomba, A. Panichella, A. De Lucia, R. Oliveto, and A. Zaidman. 2016. A textual-based technique for Smell Detection. In *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*. 1–10. https://doi.org/10.1109/ICPC.2016.7503704
[49] Y. Pan, T. Mei, T. Yao, H. Li, and Y. Rui. 2016. Jointly Modeling Embedding and Translation to Bridge Video and Language. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 4594–4602. https://doi.org/10.1109/CVPR.2016.497
[50] S. Ren, K. He, R. Girshick, and J. Sun. 2017. Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 39, 6 (June 2017), 1137–1149. https://doi.org/10.1109/TPAMI.2016.2577031
[51] V. Sales, R. Terra, L.F. Miranda, and M.T. Valente. 2013. Recommending Move Method refactorings using dependency sets. In *Reverse Engineering (WCRE), 2013 20th Working Conference on*. 232–241. https://doi.org/10.1109/WCRE.2013.6671298
[52] Olaf Seng, Johannes Stammel, and David Burkhart. 2006. 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*. 1909–1916.
[53] Tushar Sharma and Diomidis Spinellis. 2018. A survey on software smells. *Journal of Systems and Software* 138 (2018), 158 – 173.

[54] F. Simon, F.Steinbrucker, and C.Lewerentz. 2001. Metrics Based Refactoring. In *Proceedings of Europen Conference on Software Maintenance and Reengineering*. 30–38.

[55] Ricardo Terra, Marco Tulio Valente, Sergio Miranda, and Vitor Sales. 2018. JMove: A novel heuristic and tool to detect move method refactoring opportunities. *Journal of Systems and Software* 138 (2018), 19 – 36. https://doi.org/10.1016/j.jss.2017.11.073

[56] Frank Tip, Adam Kiezun, and Dirk Baeumer. 2003. 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, 13–26.

[57] Nikolaos Tsantalis and Alexander Chatzigeorgiou. 2009. Identification of Move Method Refactoring Opportunities. *IEEE Transactions on Software Engineering* 35, 3 (2009), 347–367. https://doi.org/10.1109/TSE.2009.1

[58] Stephane Vaucher, Foutse Khomh, Naouel Moha, and Yann-Gaël Guéhéneuc. 2009. Tracking design smells: Lessons from a study of god classes. In *Reverse Engineering, 2009. WCRE'09. 16th Working Conference on*. IEEE, 145–154.

[59] Weka. [n. d.]. http://www.cs.waikato.ac.nz/ml/weka/.

[60] D. Wu, N. Sharma, and M. Blumenstein. 2017. Recent advances in video-based human action recognition using deep learning: A review. In *2017 International Joint Conference on Neural Networks (IJCNN)*. 2865–2872. https://doi.org/10.1109/IJCNN.2017.7966210

[61] Zhifeng Xu. 2018. Source Code. Retrieved July 21, 2018 from https://github.com/liuhuigmail/FeatureEnvy/blob/master/Algorithm/train-CNN.py

[62] K. Zhang, W. Zuo, Y. Chen, D. Meng, and L. Zhang. 2017. Beyond a Gaussian Denoiser: Residual Learning of Deep CNN for Image Denoising. *IEEE Transactions on Image Processing* 26, 7 (July 2017), 3142–3155. https://doi.org/10.1109/TIP.2017.2662206

[63] Min Zhang, Tracy Hall, and Nathan Baddoo. 2011. Code Bad Smells: a review of current knowledge. *Journal of Software Maintenance and Evolution: Research and Practice* 23, 3 (2011), 179–202. https://doi.org/10.1002/smr.521