Deep Learning Based Feature Envy Detection Boosted by Real-World Examples

Bo Liu
Beijing Institute of Technology, China
liubo@bit.edu.cn

Hui Liu*
Beijing Institute of Technology, China
liuhui08@bit.edu.cn

Guangjie Li
National Innovation Institute of Defense Technology, China
liguangjie_er@126.com

Nan Niu
University of Cincinnati, USA
nan.niu@uc.edu

Zimao Xu
Beijing Institute of Technology, China
zimaoxu@bit.edu.cn

Yifan Wang
Huawei Cloud, China
wangyifan14@huawei.com

Yunni Xia
Chongqing University, China
xiayunni@hotmail.com

Yuxia Zhang
Beijing Institute of Technology, China
yuxiazh@bit.edu.cn

Yanjie Jiang*
Beijing Institute of Technology, China
Peking University, China
yanjiejiang@bit.edu.cn

ABSTRACT

Feature envy is one of the well-recognized code smells that should be removed by software refactoring. A major challenge in feature envy detection is that traditional approaches are less accurate whereas deep learning-based approaches are suffering from the lack of high-quality large-scale training data. Although existing refactoring detection tools could be employed to discover real-world feature envy examples, the noise (i.e., non-feature envy) within the resulting data could significantly influence the quality of the training data as well as the performance of the models trained on the data. To this end, in this paper, we propose a sequence of heuristic rules and a decision tree-based classifier to filter out non-feature envy reported by state-of-the-art refactoring detection tools. The data after filtering serve as the positive items in the requested training data. From the same subject projects, we randomly select methods that are different from positive items as negative items. With the real-world examples (both positive and negative examples), we design and train a deep learning-based binary model to predict whether a given method should be moved to a potential target class. Different from existing models, it leverages additional features, i.e., coupling between methods and classes (*CBMC*) and the message passing coupling between methods and classes (*MCMC*) that have not yet been exploited by existing approaches. Our evaluation results on real-world open-source projects suggest that the proposed approach substantially outperforms the state of the art in feature envy detection, improving precision and recall by 38.5% and 20.8%, respectively.

*Corresponding authors

ESEC/FSE '23, December 3–9, 2023, San Francisco, CA, USA

ACM ISBN 979-8-4007-0327-0/23/12…$15.00
https://doi.org/10.1145/3611643.3616353

CCS CONCEPTS

• **Software and its engineering → Software maintenance tools**; **Software evolution**.

KEYWORDS

Software Refactoring, Feature Envy, Code Smells

ACM Reference Format:
Bo Liu, Hui Liu, Guangjie Li, Nan Niu, Zimao Xu, Yifan Wang, Yunni Xia, Yuxia Zhang, and Yanjie Jiang. 2023. Deep Learning Based Feature Envy Detection Boosted by Real-World Examples. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '23), December 3–9, 2023, San Francisco, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3611643.3616353

1 INTRODUCTION

The term *code smells* was coined by Beck and Fowler [23] to represent bad designs in source code that may reduce the readability and maintainability of software projects. Different categories of code smell, e.g, *duplicated code*, *lazy class*, and *long method* have been proposed and intensively studied in both industry and academic community [15, 20]. Considering the prevalence and impact of code smells, hundreds of automated or semi-automated approaches have been proposed to identify and resolve such code smells [3, 18]. Code smells also widely serve as a useful indicator of software quality [70] and pilot lamps for software refactorings [69].

Feature envy is one of the most well-known and well-studied code smells [17, 49]. Methods associated with feature envy smells are often called *feature envy methods* or *misplaced methods* [4, 10]. Such methods are more interested in (features of) other classes than their enclosing classes, and thus they should be moved from their enclosing classes to those classes that they are interested in. The movement is known as *move method refactorings* [46]. Feature envy methods often result in unnecessary coupling between classes, increasing the difficulty of maintenance [62]. To this end, researchers have proposed dozens of approaches/tools to detect feature envy methods automatically [50, 52] and to recommend solutions (i.e., *move method refactorings*) [5, 66].

The most simple and intuitive way to identify feature envy methods is to manually design a sequence of heuristic rules and to detect feature envy methods with such predefined rules. Well-known examples include `JDeodorant` [17] and `JMove` [64]. Although such heuristics-based approaches are simple and intuitive, it is difficult to define comprehensive and accurate heuristics and it is also difficult to find the optimal setting of the thresholds that are often indispensable for heuristic rules [25]. To this end, researchers leverage traditional machine learning techniques, like SVM [32] and decision trees [2], to automatically learn rules for feature envy detection. Although such traditional machine learning techniques have the ability to learn simple rules from small labeled datasets, complex mappings (especially nonlinear mappings) from complex features of source code to the prediction are often beyond the reach of such techniques [13, 53]. To learn such complex mappings (rules), more advanced deep learning techniques have been applied to the detection of feature envy smells [41]. Although deep learning techniques have the potential to learn complex mappings, they often request a large number of labeled high-quality training data that are difficult to obtain [58]. The size and quality of the training data can significantly influence the performance of the resulting models [33]. To this end, Liu et al. [41] collected negative items (i.e., methods not associated with feature envy smells) by randomly sampling methods in high-quality open-source projects, assuming that all such methods are well-placed and are not associated with feature envy smells. Based on the same assumption, Liu et al. [41] generated positive items (i.e., feature envy methods) by randomly moving methods within high-quality open-source projects, and the resulting methods (after move method refactorings) envy their original enclosing classes (where they were placed before move method refactorings), and thus they could be taken as feature envy methods. Although this novel approach to generating training data has successfully increased the size of training datasets, it also raises serious concerns about the quality of the generated data: The arbitrarily and intentionally created feature envy smells could be essentially different from those in real-world projects that are created organically by developers. As a conclusion, the challenge in feature envy detection is that traditional approaches are less accurate whereas deep learning-based approaches are suffering from the lack of high-quality large-scale training data.

In this paper, we boost deep learning-based feature envy detection by real-world examples. Although existing refactoring detection tools, e.g., `RefactoringMiner` [67], could be employed to discover real-world *move method refactoring* examples, the noise (i.e., non-feature envy) within the resulting data could significantly influence the quality of the training data as well as the performance of the models trained on the data. The evaluation results in Section 4 confirm that directly employing the output of `RefactoringMiner` as training data would result in a substantial reduction in performance. To this end, we propose a sequence of heuristic rules and a decision tree-based classifier to filter out *potential move method refactorings* that are not associated with feature envy smells.

Another contribution of the paper is that we design a new deep learning-based model by leveraging features, i.e., *coupling between methods and classes* (*CBMC*) and *message passing coupling between methods and classes* (*MCMC*) that have not yet been exploited by existing approaches for feature envy detection. The third contribution is that we design and leverage a sequence of heuristics rules besides the deep learning-based classifier to make the final decisions in feature envy detection. We evaluate the proposed approach on five real-world open-source projects. Our evaluation results suggest that the proposed approach substantially improves the state of the art, improving precision and recall by 38.5% and 20.8%, respectively.

The paper makes the following contributions:

- An automated approach to collecting real-world feature envy examples, and a publicly available large-scale high-quality dataset of real-world feature envy methods.
- A deep learning-based approach (called `feTruth`) to detecting and resolving feature envy smells, leveraging new features not yet employed for this task.
- An initial evaluation of the proposed approach.

2 RELATED WORK

2.1 Heuristics-Based Approaches

Structural information of source code, especially dependencies, is widely used in heuristics-based detection of feature envy smells. For example, Tsantalis and Chatzigeorgiou [66] proposed a distance-based approach to identify move method refactoring opportunities and integrated the approach into the well-known refactoring tool `JDeodorant` [17]. They redefined the *Jaccard distance* [61] to measure the distance between methods and classes. If a method is closer to a class than its enclosing class, and it satisfies a set of preconditions that enable a legal move method refactoring, the method should be moved. Sales et al [56, 64] proposed a similarity-based approach `JMove` to suggest move method refactorings. They computed the average similarity between a method and a class according to their dependencies. If a method is more similar to a class than its enclosing class, it should be moved. Mayvan et al. [45] proposed a metric-based approach to identify feature envy smells. They summarized code metrics that have already been exploited by existing approaches for feature envy detection. Based on the resulting metrics, they defined the formal specification of feature envy smells and detected smells according to the metrics-based specification.

Textual information is also useful for feature envy detection because the semantic information embedded in identifiers (e.g., method names and class names) represents the roles of software entities (e.g., methods and classes) and such roles are critical factors for feature envy detection. For example, Bavota et al. [5] leveraged *Relational Topic Models* (RTM) to recommend move method refactoring opportunities by exploiting both textual information and structural information to derive semantic and structural relationships between methods. Based on such relationships, the approach (called `Methodbook`) identifies feature envy smells and suggests destinations for the feature envy methods. Palomba et al. [52] proposed a text-based approach to detecting code smells (including feature envy smells). They extracted textual contents (i.e., identifiers and comments) and normalized them by a typical information retrieval normalization process. The normalized textual content was employed to compute the textual similarity between methods and classes. If a method is more similar to a class than its enclosing class, it should be moved.

Evolution histories and refactoring histories could be exploited for the detection of feature envy smells. For example, Palomba et

al. [50, 51] exploited change histories to detect feature envy smells. They assumed that a method affected by feature envy smells should change more frequently together with the envied class than with the class where it is actually placed. Consequently, if a method changes more frequently together with a class than with its enclosing class, the method is a feature envy method. Liu et al. [42] proposed an approach to identify move method refactoring opportunities based on refactoring histories. Once developers move method m out of class C, the approach recommends moving such methods in C that have the strongest relationship and greatest similarity with m.

2.2 Machine Learning-Based Approaches

Kreimer [36] employed decision tree-based classification to detect design flaws (including feature envy smells). Fontana et al. [19, 21, 22] conducted an empirical study comparing the performance of different machine learning techniques in detecting code smells. Their evaluation results suggest that machine learning techniques can significantly improve the performance of code smell detection. Similarly, Amorim et al. [2] evaluated the performance of decision tree-based detection of code smells. Their experiment results suggest that decision trees result in the best performance.

Nucci et al. [13] replicated the study by Fontana et al. [19] with a more realistic dataset. However, their evaluation results revealed that the high performance achieved in the previous study [19] was due to the specific characteristics (e.g., unrealistically balanced dataset and biased validation methods) of the selected limited dataset. In contrast, the effect of traditional machine learning techniques in code smell detection is limited.

2.3 Deep Learning-Based Approaches

Liu et al. [41, 43] are the first to apply deep learning techniques (more specifically, CNN and fully connected networks) to the detection and resolution of feature envy smells. They exploited textual information and code metrics (i.e., the distance between methods and classes [66]) as input to a neural network-based classifier whose output indicates whether a given method should be moved from its enclosing class to another given class. They realized that existing datasets are too small for deep learning-based techniques, and thus they proposed a novel approach to create large-scale labeled datasets automatically by randomly moving methods in high-quality open-source projects. Their evaluation results suggest that the deep learning-based approach significantly outperforms the state of the art in detecting feature envy smells and in recommending solutions for feature envy methods. Barbez et al. [4] applied multi-layer perceptions (i.e., fully connected feed forward neural networks) to the detection of feature envy smells. Besides CNN and fully connected networks, other deep-learning techniques have also been applied to the detection and resolution of feature envy smells. Hadj-Kacem and Bouassida [27] adopted *coding criterion* [54] and a variational autoencoder to extract the semantic features hidden in the source code, which are fed to a logistic regression classifier to detect feature envy smells. Kurbatova et al. [37] exploited *code2vec* [1] to compute the similarity between software entities, which improves the performance of feature envy detection.

Cui et al. [9] proposed an approach to recommend move method refactoring opportunities named `RMove` by exploiting structural and semantic representations of code snippets. They investigated the performance of various code embedding and graph embedding techniques and different machine/deep learning classifiers in recommending move method refactorings. Sharma et al. [58] conducted an empirical study on deep learning-based detection of code smells. They found that CNN, RNN, and Autoencoder have been widely used in this field, and thus compared the performance of these deep learning techniques in detecting feature envy smells.

The proposed approach differs from such approaches introduced in the preceding paragraphs in that it automatically collects large-scale real-world examples as training data to boost the detection and resolution of feature envy smells. Besides that, it also exploits new features, i.e., *CBMC* and *MCMC*, that have not yet been exploited by existing approaches.

2.4 Discovering Applied Software Refactorings

Automated discovery of software refactorings is to uncover refactorings applied by developers in real-world projects. The benefits of discovering such refactorings are twofold. First, it facilitates the understanding of software evolution, i.e., what kind of changes have been made and the rationale for the changes. Second, the discovered refactorings may serve as a benchmark for the evaluation of approaches in refactoring recommendations. To this end, Dig et al. [14] proposed the first approach (called `RefactoringCrawler`) to discover software refactorings. It exploits *Shingles encoding* [7] to match software entities between two successive versions of the same project and identifies moved/deleted/added/modified/untouched software entities. It then leverages pre-defined heuristic rules to identify refactorings based on the matched software entities. For example, if a method m in the older version matches method m' in the later version but their enclosing classes (noted as classes A and B) do not match, `RefactoringCrawler` would report that there is a move method refactoring moving method m from its enclosing class A to class B.

Prete et al. [35, 55] proposed a logic programming-based approach, called `Ref-Finder`, to discover software refactorings. `Ref-Finder` encodes refactoring types into template logic rules and extracts logic facts concerning two successive versions into a logic database. A logic programming engine [11] is then employed to identify refactorings by converting the logic rules into logic queries and executing the queries on the logic database. `RefDiff` proposed by Silva et al. [59, 60] is a similarity-based approach to discovering software refactorings. It differs from other approaches in that it computes the similarity between software entities (i.e., two successive versions) with a variation of the TF-IDF weighting scheme [57] and a weighted Jaccard coefficient [8].

`RefactoringMiner`, proposed by Tsantalis et al. [67, 68], is a state-of-the-art approach to discovering refactorings. The key to this approach is an AST-based statement matching algorithm that does not request any user-defined thresholds. The algorithm accurately and automatically matches software entities between two successive versions. Based on the matched entities, `RefactoringMiner` exploits pre-defined rules (similar to those used by `RefactoringCrawler` [14]) to detect refactorings. Their evaluation results suggest that `RefactoringMiner` can significantly improve the state of the art in refactoring detection.

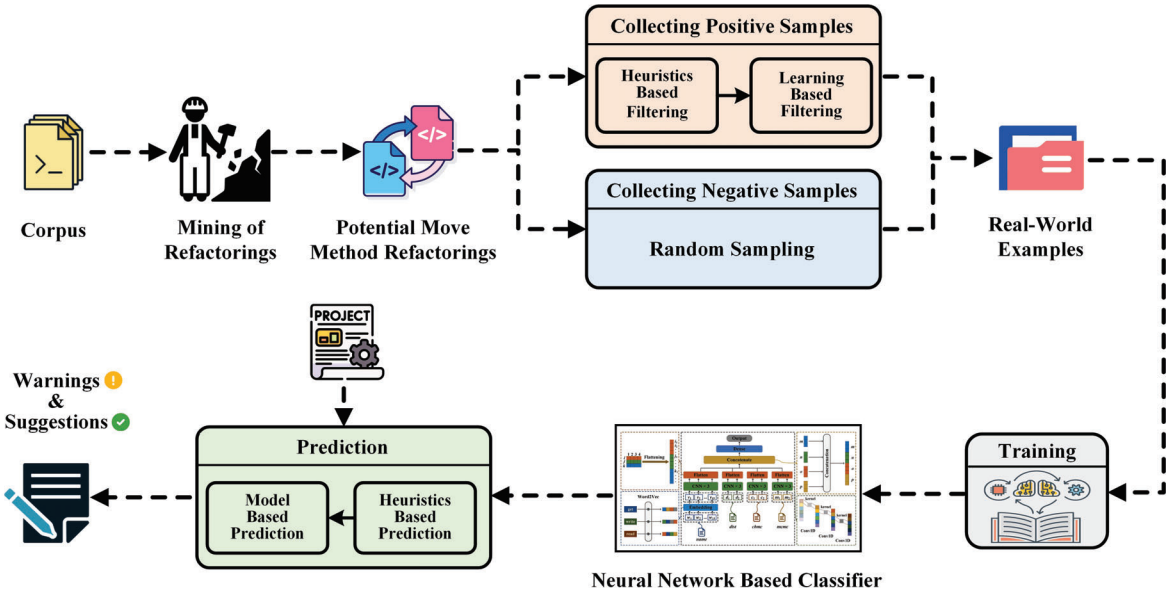

Figure 1: Overview of the Proposed Approach

3 APPROACH

3.1 Overview

An overview of the proposed approach is presented in Fig. 1. For convenience, we call the proposed approach `feTruth` where "`fe`" is the abbreviation of "`feature envy`" and "`Truth`" suggests that the approach is boosted by the truth (i.e., real-world examples) and it could tell you the truth of feature envy.

`feTruth` takes as input the evolution histories of open-source projects stored in version control systems (e.g., GitHub), extracts real-world feature envy examples, and trains a deep learning-based prediction model with the extracted examples. In the testing phase, `feTruth` takes as input the source code of a software project and generates a list of feature envy smells associated with methods in the project, as well as suggested refactoring solutions to resolve the feature envy smells. The overall process of `feTruth` is as follows:

- `feTruth` mines for *potential move method refactorings* by applying `RefactoringMiner` [67] to the evolution histories of open-source projects.
- `feTruth` removes non-feature envy from the potential refactorings by heuristics-based filtering and learning-based filtering. The methods moved by the remaining *move method refactorings* are taken as positive samples.
- Methods not involved in any potential move method refactorings are taken as negative samples.
- With such positive and negative samples collected in the preceding steps, `feTruth` trains a neural network-based classifier to predict whether a given method should be moved to another class.
- `feTruth` leverages the trained classifier as well as a sequence of heuristic rules to predict whether a given method in the testing project is associated with *feature envy smells*. If yes, `feTruth` also suggests which class the method should be moved to.

Details of the key steps are presented in the following sections.

3.2 Mining for Move Method Refactorings

Feature envy smells are often resolved by move method refactorings [17]. Consequently, it is practical to discover methods associated with feature envy smells by mining for move method refactorings in open-source projects. To this end, we employ `RefactoringMiner` [67] to discover potential move method refactorings. `RefactoringMiner` is selected because it is well-known, widely-used, and representing the state of the art in automated discovery (mining) of software refactorings. It takes as input two versions (i.e., a commit and its parent in the commit history) of the same project, compares and matches software entities between the two versions, and generates a list of potential move method refactorings (as well as other categories of refactorings that are ignored by `feTruth`) by a sequence of pre-defined detection rules.

Although `RefactoringMiner` represents the state of the art, not all of the identified move method refactorings are associated with feature envy smells, and it also results in some false positives (i.e., the actual changes made are not move method refactorings). According to our empirical study in Section 4.5, up to 43.8% of the reported potential move method refactorings are not associated with feature envy smells or are false positives. For convenience, we call such potential refactorings as *non-feature envy* in the rest of this paper. To guarantee the quality of collected real-world examples, `feTruth` leverages heuristics-based filtering and classification-based filtering to identify and remove such non-feature envy. Details of the filtering are presented in Section 3.3 and Section 3.4, respectively.

3.3 Heuristics-Based Filtering

The proposed approach is to identify feature envy methods that should be moved from their enclosing classes to other classes

that they envy. Consequently, for a given feature envy method, it should explicitly have its enclosing class (where it is) and its target class (where it should be moved) in the same version. However, `RefactoringMiner` often reports potential move method refactoring where the target class of the movement does not exist in the old version (i.e., V_n) or the source class does not exist in the new version (i.e., V_{n+1}). A typical example is that when a class is moved to a different package, `RefactoringMiner` may report methods within the class as *moved methods* where the class in the old version is noted as the source class and the same class in the new version is noted as the target class. In this case, the target class does not exist in the old version. Consequently, it is unlikely for feature envy detection algorithms (including the proposed approach) to identify them as feature envy methods because the algorithms cannot recommend moving a method to a nonexistent class. Another typical example is that when a class (noted as A) is merged into another class (noted as B), `RefactoringMiner` may report that the methods within A have been moved to the merged class (i.e., class B). In this case, the source class (i.e., class A) of the *move method refactorings* do not exist in the new version (V_{n+1}). Such detection is not appropriate as feature envy examples because it is a *merge class* refactoring or *inline class* refactoring, instead of a sequence of move method refactorings. Therefore, these potential move method refactorings should not be part of the training data because they are not associated with feature envy smells.

To exclude such potentital refactorings, we exclude a potential move method refactoring if:

(1) The source class of the potential refactoring does not exist in the new version (we note this heuristic rule as H_1) or
(2) The target class of the potential refactoring does not exist in the old version (we note this heuristic rule as H_2).

Besides the heuristic filtering rules proposed in the preceding paragraph, we also leverage another rule (noted as H_3) to exclude testing methods, constructors, overriding methods, and overridden methods: *If the method is moved by a potential move method refactoring is a testing method, a constructor, an overriding method or an overridden method, the method should not be taken as a feature envy method.* Testing methods are excluded because the proposed approach is specially designed for production code. Notably, testing methods are often substantially different from methods in production code. A testing method, located in a testing class, frequently calls methods and accesses fields from the class (noted as `ProductClass`) it is testing. As a result, it inherently envies the `ProductClass`, and it may be misclassified by feature envy detection tools as a feature envy method. To this end, the proposed approach, as well as other existing approaches (e.g., `feDeep` proposed by Liu et al [41]), excludes testing methods. Constructors are excluded because they cannot be moved at all. Overriding and overridden methods are excluded because they cannot be moved across inheritance hierarchies, and moving between ancestors and descendants are often taken as *pull up (or push down) method refactorings* instead of *move method refactorings*. Pull up (and push down) method refactorings are designed to share (or conceal) functionality and interfaces among sibling classes, instead of removing feature envy smells. Consequently, the involved methods are often not associated with feature envy smells.

3.4 Learning-Based Filtering

Besides the heuristics introduced in the previous section, we also leverage a learning-based approach to further filter out non-feature envy. More specifically, we leverage a decision tree-based classifier to distinguish false positives from true positives according to a sequence of features of the potential move method refactorings.

A potential move method refactoring is represented as:

$$pMTr =< m, m', sc, tc > \tag{1}$$

where m and m' refer to the method before and after refactoring whereas sc and tc represent the source class and the target class of the movement. Methods calling m and m' are noted as $Caller(m)$ and $Caller(m')$, respectively.

The exploited features of a potential move method refactoring are explained as follows:

(1) rsc: The ratio of callers to the original method that survived the movement, i.e.,
$$rsc = \frac{|Caller(m) \cap Caller(m')|}{|Caller(m)|} \tag{2}$$
A caller to the original method m survives the movement if and only if it calls m' after the movement.
(2) rsc': The number of survived callers divided by the number of callers to m', i.e.,
$$rsc' = \frac{|Caller(m) \cap Caller(m')|}{|Caller(m')|} \tag{3}$$
(3) cst: The number of common statements (matched statements according to `RefactoringMiner`) appearing in both m and m', i.e.,
$$cst = |Statements(m) \cap Statements(m')| \tag{4}$$
where $Statements(m)$ represents all statements within method m.
(4) $rcst$: The number of common statements divided by the total number of statements in m, i.e.,
$$rcst = \frac{cst}{|Statements(m)|} \tag{5}$$
(5) $rcst'$: The number of common statements divided by the total number of statements in m', i.e.,
$$rcst' = \frac{cst}{|Statements(m')|} \tag{6}$$

Features rsc and rsc' concern to what extent m and m' share the same callers whereas features cst, $rcst$, and $rcst'$ concern the similarity between the method bodies of m and m'. Such outside features (concerning the callers to the methods) and inside features (concerning the method bodies) together may help identify whether m' is an evolved version of m. Notably, most of the false positives reported by `RefactoringMiner` are caused by incorrect matching between m and m': m' is somewhat similar to m (which results in incorrect matching), but m' is not an *evolved version* of m, and thus they do not form a move method refactoring.

With the extracted features, we train a decision tree to learn how to classify potential refactorings into false positives and true positives. It takes $< rsc, rsc', cst, rcst, rcst' >$ as input, and generates a binary output that suggests whether the given potential move method refactoring is associated with feature envy smells. Decision trees [6] are one of the most popular and commonly used machine

learning techniques for classification [63]. Compared to other machine learning techniques, decision trees are intuitive, interpretable, and adjustable (via manual pruning to optimize classification). We do not employ more powerful deep learning techniques because they request a large number of labeled training data that we do not have. In contrast, decision trees work well on small datasets.

3.5 Collecting Negative Samples

In the previous sections, we collect real-world samples of feature envy methods with `RefactoringMiner` and a sequence of filters as introduced in Sections 3.3–3.4. In this section, we explain how we collect negative samples, i.e., methods not associated with feature envy smells. Such negative examples together with the positive samples constitute the complete labeled training data for feature envy detection.

We first collect all methods in subject projects where positive samples are collected. From such methods, we exclude the following methods:

- Methods that cannot be moved across inheritance hierarchies, e.g., constructors, overriding methods, and overridden methods;
- Testing methods that are out of the scope of the proposed approach;
- Methods that violate move method refactoring preconditions provided by `Eclipse JDT` [28, 29] (e.g., the target class should not inherit a method having the same signature of the moved method);
- Methods that have been moved by potential *move method refactorings* according to the detection results of `RefactoringMiner`.

The remaining methods are noted as *candidate negative samples*. Considering that in most high-quality projects negative samples are often significantly more popular than positive samples (feature envy methods), we conduct undersampling [44] so that the total number of negative examples equals the number of positive samples. To maximize the diversity of the negative examples, we sample no more than a single method from each class and keep the same sample rate on all subject projects.

3.6 Detection and Resolution of Feature Envy

First, the following methods are automatically predicted as negative (i.e., methods not associated with feature envy smells): Constructors, overriding methods, and overridden methods. These methods are predicted as negative because they cannot be moved across inheritance hierarchies. For other methods, we leverage a neural network-based classifier for automated prediction. An overview of the classifier is presented in Fig. 2 and its details are discussed as followings.

3.6.1 Input of the Classifier. The classifier takes as input a method (noted as m) and a potential target class (noted as C_t) to which the method could be legally moved. It generates a binary prediction to suggest whether m should be moved to C_t. Notably, existing tools, like `Eclipse JDT`, can automatically and accurately validate whether m could be legally moved to target class C_t. Consequently, we reuse `Eclipse JDT` to collect all potential target classes for a given method m and leverage the classifier to predict whether m should be moved to any of the potential target classes.

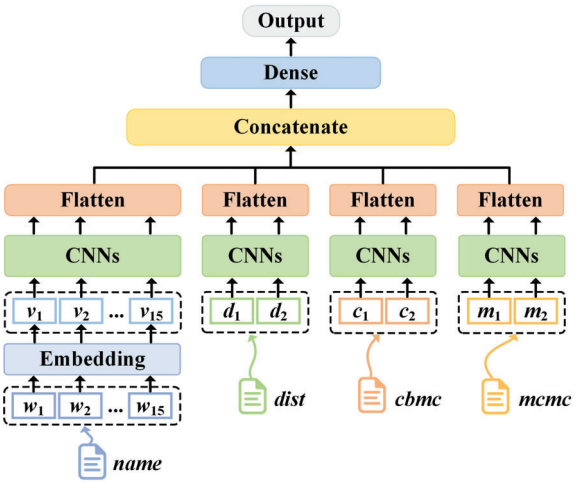

Figure 2: Neural Network Based Classifier

The input of the classifier consists of both textual features and structural features (code metrics):

$$input = < names, metrics > \tag{7}$$

where *names* consist of three identifiers:

$$names = < name(m), name(C_s), name(C_t) > \tag{8}$$

where $name(m)$, $name(C_s)$, $name(C_t)$ represent the name of the method to be tested, the name of its enclosing class, and the name of the potential target class. Structural features *metrics* consist of a sequence of code metrics:

$$\begin{aligned} metrics = < & dist(m, C_s), dist(m, C_t), cbmc(m, C_s), \\ & cbmc(m, C_t), mcmc(m, C_s), mcmc(m, C_t) > \end{aligned}$$

where $dist(m, C)$ is the *Jaccard distance* between method m and class C [61, 66]. Let S_m be the entity set of method m and S_C be the entity set of class C, the distance between m and C is defined as follows:

$$dist(m, C) = 1 - \frac{|S_m \cap S_C|}{|S_m \cup S_C|} \tag{9}$$

Notably, the entity set of a software entity (e.g., a method or a class) includes all methods and fields accessed directly by the entity. Besides the distance (as defined by Equation 9) that has been exploited by existing deep learning-based approaches [41, 43], our approach also exploits *CBMC* and *MCMC*. $cbmc(m, C)$ is the number of methods from class C that are called directly by m divided by the total number of methods directly called by m:

$$cbmc(m, c) = \frac{|\texttt{methods called by } m \cap \texttt{ methods of } C|}{|\texttt{methods called by } m|} \tag{10}$$

$mcmc(m, C)$ is the frequency of invocations from method m to class C divided by the total frequency of invocations from m:

$$mcmc(m, c) = \frac{|\texttt{method invocations from } m \texttt{ to } C|}{|\texttt{method invocations from } m|} \tag{11}$$

3.6.2 Architecture of the Classifier. The architecture of the classifier is presented in Fig. 2. The textual features are first fed into an embedding layer to convert them into numerical vectors. We leverage the well-known and widely-used word2vec [47, 48] as the embedding layer, and reuse the publicly available pre-trained word2vec model [40]. The outputs of the embedding layer, together with other structural features, are fed into convolutional neural networks (CNN). The outputs of the CNN layers are transformed into one-dimensional vectors by the following flatten layers. Such vectors are merged into a single one-dimensional vector by the following merge layer, and the resulting vector is fed into a dense layer. The output layer generates the binary prediction, i.e., whether m should be moved from its enclosing class to the target class C_t.

3.6.3 Classifier-Based Prediction. For each method to be tested (noted as m), `feTruth` first leverages `Eclipse JDT` [28, 29] to collect potential target classes to which the method could be legally moved. Notably, from the potential target classes, we remove all data classes that do not contain any methods or contain getter/setter methods and constructors only (we note this heuristic rule as H_4). Such classes are specially designed to store data (variables) only, and moving methods to such classes may break the principle. Besides, if m is an instance method (i.e., a non-static method) and its a target class is composed of only static members, we remove this class from the target classes (we note this heuristic rule as H_5). Consequently, we exclude them from potential target classes.

If a method does not have any potential target class, the method is deemed negative, not associated with feature envy smells (we note this heuristic rule as H_6). Otherwise, `feTruth` leverages the neural network-based classifier represented in Fig. 2 to predict whether the method should be moved to the given potential target class. The prediction is often interpreted as *positive* (and the potential target class is called *candidate target class*) if the output is greater than 0.95 [34]. Otherwise, it is interpreted as *negative*. If the prediction is negative for all potential target classes, the method is not associated with feature envy smells. Otherwise, it is a feature envy method.

For methods predicted as positive, `feTruth` also suggests solutions, i.e., where they should be moved. If method m has only a single candidate target class, `feTruth` suggests moving the method to this class. If it has multiple candidate target classes, `feTruth` suggests moving m to the candidate target class that results in the largest output of the classifier.

4 EVALUATION

4.1 Research Questions

The evaluation was designed to answer the following research questions:

RQ1. Can `feTruth` improve the state of the art in feature envy detection?

RQ2. Are the filters proposed in Sections 3.3–3.4 accurate in excluding non-feature envy in potential move method refactorings reported by `RefactoringMiner`?

RQ3. To what extent can data filtering improve the performance of `feTruth`?

RQ4. To what extent can real-world examples outperform randomly generated training data in boosting `feTruth`?

Table 1: Subject Projects (Part2)

Projects	Snapshot	NOC	NOM	LOC
Jsoup	9b40b7b	90	1,317	9,763
Csv	989c495	15	158	2,125
Compress	231a466	81	664	8,534
Cli	7d1363e	41	343	4,460
Time	b9a83fb	317	9,290	79,036

RQ5. How does the size of training data influence the performance of `feTruth`?

Research question RQ1 concerns whether `feTruth` can substantially outperform existing approaches in feature envy detection. To answer this question, we compare `feTruth` against the approach proposed by Liu et al. [41], `JDeodorant` [17] and `JMove` [64]. Since Liu et al. [41] did not explicitly name their approach, we call it `feDeep` for convenience in the rest of this paper. Such baseline approaches are selected for comparison because they represent the state of the art. Research question RQ2 concerns the accuracy of the filtering introduced in Sections 3.3–3.4 whereas RQ3 concerns its influence on the performance of the proposed approach. RQ4 concerns the benefits of replacing randomly generated training data with real-world examples whereas RQ5 concerns the impact of the size of the training data.

4.2 Subject Projects

The subject projects are divided into two parts. The first part (noted as `Part1`), consisting of 500 Java projects, was used to discover real-world feature envy examples (i.e., collection of training data). The second part (noted as `Part2`), consisting of 5 open-source Java projects, was used to evaluate the proposed approach and the selected baseline approaches. The 500 projects in the first part were selected from GitHub: We selected the top 500 most popular Java projects (with the largest numbers of stars) to constitute the first part of the subject projects. The 5 projects in the second part were selected from Defects4J [31]. These projects were selected because they are from different domains and are developed/maintained by different teams, which may help to reduce the potential bias in the evaluation. Besides, all of them are well-known and widely-used open-source projects. Table 1 presents an overview of the 5 Java projects in the second part where the `snapshot` specifies the version of the selected projects, `NOC` and `NOM` represent the number of classes and the number of methods within the projects. `LOC` represents the number of source code lines.

4.3 Process

First, we applied `RefactoringMiner` to the 500 selected subject projects (in `Part1`), and it reported 30,599 potential move method refactorings. From them, we randomly sampled 600 potential move method refactorings, and requested three experienced developers who were familiar with move method refactorings and feature envy smells to independently and manually marked them as true positives (i.e., move method refactorings associated with feature envy smells) or non-feature envy. All of the participants were required

Table 2: Improving the State of the Art

Approach	Project	#Reported	#Accepted	#Accepted targets	Precision	Accuracy (destination)
feTruth	Jsoup	16	12	11	75%	91.67%
	Csv	3	2	2	66.67%	100%
	Compress	13	8	7	61.54%	87.5%
	Cli	1	1	1	100%	100%
	Time	8	6	6	75%	100%
	Total	41	29	27	70.73%	93.1%
feDeep	Jsoup	19	10	8	52.63%	80%
	Csv	0	0	0	0%	0%
	Compress	14	7	6	50%	85.71%
	Cli	1	1	1	100%	100%
	Time	13	6	6	46.15%	100%
	Total	47	24	21	51.06%	87.5%
JDeodorant	Jsoup	14	5	5	35.71%	100%
	Csv	1	1	1	100%	100%
	Compress	4	2	1	50%	50%
	Cli	1	1	1	100%	100%
	Time	3	1	0	33.33%	0%
	Total	23	10	8	43.48%	80%
JMove	Jsoup	5	2	2	40%	100%
	Csv	0	0	0	0%	0%
	Compress	1	1	1	100%	100%
	Cli	2	1	1	50%	100%
	Time	2	0	0	0%	0%
	Total	10	4	4	40%	100%

to have Java background. They had a median of 6 years of programming experience and 3.5 years experience with software refactoring. In case of inconsistent labeling, the potential refactorings were discussed together by the participants to reach an agreement, which resulted in 600 consistently labeled samples. Notably, the three developers achieved high consistency with a Fleiss' kappa coefficient [16] of 0.81. The size of the sample (600) guaranteed a confidence level of over 95% and a margin of error of 5% [30].

Second, we trained the decision tree-based filter (as presented in Section 3.4) with the 600 manually labeled samples. After that, we leveraged the resulting decision tree-based filter and heuristics-based filter (as presented in Section 3.3) to identify all positive examples reported by `RefactoringMiner`, which resulted in 14,209 real-world feature envy methods. We also collected 14,209 negative examples as introduced in Section 3.5 to make up a balanced training dataset (called *rw-Dataset*).

Third, we trained the proposed approach (`feTruth`) with the training data collected in the preceding paragraph. We trained the baseline approach (`feDeep`) with equally sized training data (called *rg-Dataset*), where positive examples were randomly generated by `feDeep` [41] on the same subject projects, and negative examples were consistent with those in *rw-Dataset*. After that, we applied the resulting models and other selected baseline approaches, i.e., `JDeodorant` [17] and `JMove` [64] to the five projects in `Part2` and requested three experienced developers to independently and manually validate all of the reported feature envy smells. To reduce the threats to validity, the participants did not know which of the feature envy smells were reported by the proposed approach or the baseline approaches. In case of inconsistency, they were requested to discuss together, and they reached an agreement on all items. Based on the manual validation, we computed the performance of the evaluated approaches.

4.4 RQ1: Improving the State of the Art

The evaluation results are presented in Table 2. `#Reported` presents the total number of potential feature envy smells reported by the evaluated approaches whereas `#Accepted` presents how many of them were confirmed according to the given ground truth. `#Accepted targets` presents how many of the target classes recommended by the approaches were confirmed. `Precision` presents the precision in smell detection. `Accuracy (destination)` presents the accuracy of the approaches in recommending target classes for the confirmed feature envy methods.

From Table 2 we make the following observations:

- `feTruh` was accurate in detecting feature envy smells. Among the 41 items reported by `feTruh`, 29 have been manually confirmed, resulting in a precision of 70.73%. `feTruh` was also accurate in suggesting target classes for feature envy methods. For 29 feature envy methods, it succeeded in recommending target classes for 27 of them, resulting in a precision of 93.1%.
- `feTruh` substantially outperformed the state of the art in feature envy detection. It resulted in the highest precision (70.73%),

```
1  private void populateProviderWithExtraProps(PoolingConnectionProvider cp,
       Properties props) throws Exception {
2      Properties copyProps = new Properties();
3      copyProps.putAll(props);
4      ...
5      if (cp instanceof C3p0PoolingConnectionProvider) {
6          copyProps.remove(C3p0PoolingConnectionProvider.
               DB_MAX_CACHED_STATEMENTS_PER_CONNECTION);
7          copyProps.remove(C3p0PoolingConnectionProvider.
               DB_VALIDATE_ON_CHECKOUT);
8          copyProps.remove(C3p0PoolingConnectionProvider.
               DB_IDLE_VALIDATION_SECONDS);
9          copyProps.remove(C3p0PoolingConnectionProvider.
               DB_DISCARD_IDLE_CONNECTIONS_SECONDS);
10     }
11     setBeanProps(cp.getDataSource(), copyProps);
12 }
```

Listing 1: Example False Positive Reported by feDeep

substantially higher than that of feDeep (51.06%), JDeodorant (43.48%), and JMove (40%). The minimal improvement is 38.5% (70.73%-51.06%)/51.06%. feTruh also led to the largest number (29) of accepted items (i.e., true positives), substantially larger than that of feDeep (24), JDeodorant (10) and JMove (4).

- The proposed approach feTruh is accurate in suggesting destination classes for feature envy methods. Its accuracy (93.1%) is higher than that of JDeodorant (80%), and comparable to that of feDeep (87.5%) and JMove (100%).

Note that all of the evaluated approaches were evaluated on the same dataset, and thus they should have the same number of positives (i.e., the total number of feature envy smells). As a result, the improvement in the number of accepted items (#true positives) equals the improvement in recall because recall = #true positives ÷ #positives. Consequently, compared to feDeep, JDeodorant, and JMove, feTruh improved the recall by 20.8%=(29-24)/24, 190%=(29-10)/10, and 625%=(29-4)/4, respectively.

We conclude based on the preceding analysis that feTruth substantially improved the state of the art in detecting and resolving feature envy smells. In the following paragraphs, we explain with examples why feTruth can outperform baseline approaches.

The example code in Listing 1 explains why feTruh succeeded in avoiding some false positives reported by feDeep. The method *populateProviderWithExtraProps* is from the class *StdSchedulerFactory* in open-source project quartz [12]. feDeep reported it as a feature envy method because it accesses four fields of class *C3p0PoolingConnectionProvider* (Lines 6-9), and thus it suggested moving it to class *C3p0PoolingConnectionProvider*. However, original developers refused to move the method. The relationship between class *StdSchedulerFactory* and class *C3p0PoolingConnectionProvider* follows the widely-used factory pattern [24] where the factory is responsible for creating objects whereas providers are responsible for providing requested data. The involved method *populateProviderWithExtraProps* initializes fields of the created object with data from providers (including the target class *C3p0PoolingConnectionProvider* and other providers). Consequently, the current decision well follows the widely-used factory pattern and there is no need to move the method. Besides, the target class *C3p0PoolingConnectionProvider* is designed to provide static fields only, and it should not contain complex operations on such fields.

```
1  boolean isEndOfFile(int c) {
2      return c == ExtendedBufferedReader.END_OF_STREAM;
3  }
4
5  class ExtendedBufferedReader extends BufferedReader {
6      static final int END_OF_STREAM = -1;
7      ...
```

Listing 2: Example True Positive Missed by Baseline

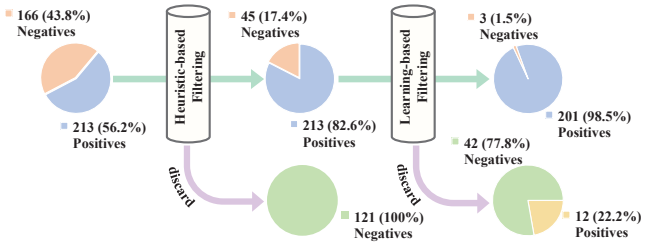

Figure 3: Filtering of Potential Move Method Refactorings

Our approach feTruth did not report the method in Listing 1 as a feature envy smell because its training data did not contain any real-world feature envy methods that should be moved from a factory class ("*Factory") to a provider class ("*Provider"). It may suggest that methods are rarely misplaced (by developers) between factories and providers. However, the training data automatically generated by feDeep did contain four artificial 'feature envy methods' that should be moved from factories to providers. Such unreal training data resulted in incorrect prediction on the method in Listing 1.

The example code in Listing 2 explains why feTruth succeeded in identifying some true positives missed by the baseline approaches. The method *isEndOfFile* is from the class *Lexer* in open-source project commons-csv [26]. feTruth reported the method in Listing 2 as a feature envy smell because its training data contained common patterns, i.e., a boolean method ("isEndOf*") was moved to the class to which the field accessed in its conditional expression belongs [65]. However, the training date generated randomly by feDeep did not contain such patterns. Notably, the heuristic-based baseline approaches also missed the method presented in Listing 2.

4.5 RQ2: Accurate Filtering of Move Methods

To answer RQ2, we randomly sampled and validated 379 (out of 29,999) items that have been fed into the filters. The size of the sample guaranteed a confidence level of 95% and a margin of error of 5% [30]. The manual checking was conducted by three experienced developers in the same way as they labeled training data (as introduced in the first paragraph of Section 4.3). They achieved high consistency with a Fleiss' kappa coefficient [16] of 0.87. Notably, while labeling the items, the participants did not know the rationale of the filters or the results of the filters, which might help to reduce potential bias. The manually created labels served as the ground truth for the evaluation of the accuracy of the filters.

The evaluation process and evaluation results are plotted in Fig. 3 where negative samples are those that were reported by RefactoringMiner as potential move method refactorings but were denied manually by the participants. Such negative samples

Table 3: Performance of `feTruth` with/without Data Filtering

Metrics	`Without Filtering`	`With Filtering`
#Reported	52	41
#Accepted	28	29
#Accepted targets	26	27
Precision	53.85%	70.73%
Accuracy (destination)	92.86%	93.1%

Table 4: Effect of Real-World Examples

Metrics	`Randomly Generated Data`	`Real-World Examples`
#Reported	49	41
#Accepted	25	29
#Accepted targets	22	27
Precision	51.02%	70.73%
Accuracy (destination)	88%	93.1%

should be identified and filtered out by the proposed filters. In contrast, the positive samples are those that were reported by `RefactoringMiner` as potential move method refactorings and were confirmed manually by the participants. Ideally, all of the positive samples should pass the filters.

From Fig. 3 we make the following observations:

- First, a large percentage of the potential move method refactorings reported by `RefactoringMiner` was not associated with feature envy smells. The negative items account for 43.8%=166/(166+213) of the reported samples.
- Second, the heuristics-based filter was effective and accurate. It successfully identified 121 out of the 213 negative items whereas none of the 166 positive items were filtered out by mistake.
- Third, the decision tree-based filter was accurate as well. It filtered out 42 out of the 45 negative items fed into the filter whereas only 12 out of the 213 positive items were misclassified.
- Finally, the two filters together filtered out 98.2%=(121+42)/166 of the negative samples whereas only 5.6%=(0+12)/213 of the positive examples were filtered out by mistake.

Based on the preceding analysis, we conclude that the filters proposed in this paper have the potential to filter out most of the negative samples accurately without significant loss in positive samples. Consequently, with `RefactoringMiner` and such filters, it is practical to construct a large-scale and high-quality dataset of feature envy smells, which in turn may boost learning-based approaches to detecting and resolving feature envy smells.

4.6 RQ3: Impact of Example Filtering

To evaluate the effect of example filtering, we disabled the filtering and repeated the evaluation (keeping the testing data unchanged). The evaluation results are presented in Table 3.

From Table 3 we make the following observations:

- The filtering had a substantial and positive impact on the performance in detecting feature envy smells. Enabling it improved the precision by 31.3%=(70.73%-53.85%)/53.85%. The number of true positives (#Accepted) also increased slightly from 28 to 29.
- The filtering substantially reduced the number of false positives (i.e., #Reported - #Accepted) from 24=52-28 to 12=41-29, with a substantial reduction of 100%=(24-12)/24.
- The filtering also had a positive impact on the accuracy of the approach in suggesting destinations for feature envy methods. The accuracy was slightly improved from 92.86% to 93.1%.

We conclude from the preceding analysis that simply employing the refactoring histories from refactoring miners without essential filtering may result in a substantial reduction in the performance of feature envy detection. This reduction stems primarily from the failure to filter out move method refactorings that are not associated with feature envy smells and are false positives. Consequently, it becomes evident that the unfiltered move method refactorings not only introduce noise into the training dataset but also hamper the model's ability to generalize accurately, thereby reducing the performance of feature envy detection.

4.7 RQ4: Effect of Real-World Examples

To investigate the effect of real-world examples, we replaced such examples with randomly generated training data (i.e., *rg-Dataset*) and repeated the evaluation. Notably, in contrast to answering RQ1, we employ the same neural network architecture (i.e., `feTruth`) for this evaluation. Testing data were kept untouched as well. The evaluation results are presented in Table 4. From this table we make the following observations:

- Exploiting real-world examples substantially improved the precision of the proposed approach. The improvement in precision is 38.6%=(70.73%-51.02%)/51.02%.
- Exploiting real-world examples improved the number of true positives (#Accepted) from 25 to 29, with a substantial improvement of 16%=(29-25)/25.
- Replacing random examples with real-world examples improved the performance in the recommendation of target classes. The accuracy was improved from 88% to 93.1%.

We conclude that real-world examples are more effective than randomly generated training data. This conclusion arises from the observation that applying models trained on randomly generated data to real-world examples results in a substantial reduction in performance. The reduction can be attributed to the fact that randomly generated data often do not align with real-world examples. Consequently, when these models, trained on such randomly generated data, are tested on real-world examples, they tend to identify fewer instances of feature envy methods and their suggested target classes are less accurate.

4.8 RQ5: Impact of Data Size

Fig. 4 presents how the size of the training data influences the performance of the proposed approach. From this figure we make the following observations:

- The precision of the approach in both detecting feature envy smells and recommending solutions (target classes) keeps increasing with the increase in training data. It may suggest that

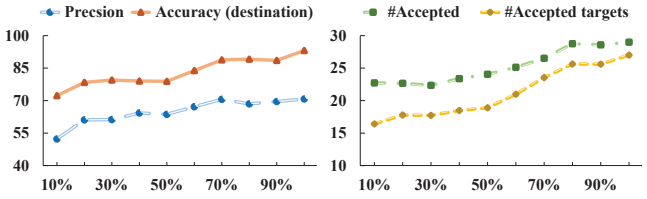

Figure 4: Performance of `feTruth` vs. Size of Training Data

the performance of the proposed approach has the potential to be further improved with additional training data in future.

- The number of confirmed feature envy smells and the number of accepted target classes keeps increasing when the data size increases. However, when the size reaches 80% of the current size, the rate of increase in the number becomes relatively slow.

Based on the analysis we conclude that collecting additional training data may further improve the proposed approach. This conclusion aligns with the underlying principle that as the training data expands, the model's performance progressively improves. This phenomenon underscores the fundamental nature of neural networks, which rely on large amounts of data for effective training.

4.9 Threats to Validity

The first threat to external validity is that only a limited number of subject projects were used for the evaluation. Only 500 subject projects were used for the mining of refactorings, and only five projects were used for testing. Note that manual validation of the detection results was tedious and time-consuming, which significantly limited the size of the testing data. To reduce the threat, we collected projects from different domains and different development teams. We also publish the implementation [38] of the proposed approach to facilitate further validation on additional subject projects.

The second threat to external validity is that the evaluation is confined to Java projects only. It is unclear how the proposed approach works on other programming languages although the proposed approach is expected to work with any object-oriented programming languages because it does not depend on any special characteristics of Java. The evaluation is confined to Java because 1) the prototype implementations of the proposed approach and the baseline approach `feDeep` are confined to Java projects only, and 2) RefactroingMiner is confined to Java projects.

A threat to construct validity is that the manual labeling of potential move method refactorings, the manual checking of reported feature envy smells, and the manual checking of suggested target classes could be inaccurate. It might in turn result in inaccurate calculation of the performance of the evaluated approaches. To reduce the threat, we requested multiple participants to label (check) the same items and computed the consistency among the participants. The resulting kappa coefficients suggest that they resulted in a high level of consistency. Besides, we did not tell them the rationale of the proposed approach, and they did not know which approaches had reported the potential smells or suggested the target classes. All of these might help reduce the bias.

5 CONCLUSIONS AND FUTURE WORK

Detection and resolution of feature envy smells have been well-studied, and dozens of such approaches have been proposed. Although deep learning techniques have been proven useful in automated detection and resolution of feature envy smells [41], the quality of the randomly generated training data is preventing them from reaching the maximal potential. To this end, in this paper, we boost deep learning-based feature envy detection approaches with real-world examples. We propose a heuristics-based filter and a learning-based filter to exclude false positives reported by refactoring miners, and manage to generate high-quality and large-scale training data for feature envy detection. We design a new deep learning-based classifier leveraging new features not yet exploited by existing approaches, and employ the resulting classifier as well as a sequence of heuristics rules to detect feature envy smells and to generate solutions for detected smells. Our evaluation results on real-world open-source projects suggest that the proposed approach substantially outperforms the state of the art in the detection and resolution of feature envy smells, improving the precision and recall in feature envy detection by 38.5% and 20.8%, respectively.

Similar to the state-of-the-art feature envy smell detection tools (e.g., `feDeep`, `JDeodorant`, and `JMove`), `feTruth` only detects misplaced methods that should be moved from their enclosing classes to other classes they envy. Although such feature envy methods are the key concern for current feature envy detection and resolution tools/algorithms, there is another category of feature envy: Only a small part of the method should be extracted and moved outside the enclosing class. We plan to generalize the proposed approach to deal with such feature envy smells in future.

`feTruth` was sponsored by Huawei, one of the leading IT companies in the world, and it has been successfully deployed in the company. Notably, we adopt a client-server architecture, putting all deep-learning related models on a server (deployed within the company) and integrating other models into a plug-in of the IDE, to avoid any deep learning computation on client sides (i.e., PCs of software developers). As a result, for the end users, `feTruth` is as simple as the traditional heuristics-based detection tools, and no deep learning libraries or devices are required. We would like to report detailed usage of the tool within the company (and potentially other companies) in the near future.

The large-scale high-quality dataset of real-world feature envy examples constructed in this paper is an important contribution of this paper. It may inspire and boost other learning-based approaches to feature envy detection in future. We also plan in future to further enlarge the dataset by mining additional evolution histories of high-quality open-source projects.

6 DATA AVAILABILITY

The replication package, including the tools and the data, is publicly available [38, 39].

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers from ESEC/FSE '23 for their insightful comments and constructive suggestions. This work was partially supported by the National Natural Science Foundation of China (62232003 and 62172037).

REFERENCES

[1] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. Code2vec: Learning Distributed Representations of Code. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 40:1–40:29. https://doi.org/10.1145/3290353

[2] 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 *Proceedings of the 26th IEEE International Symposium on Software Reliability Engineering (ISSRE '15)*. IEEE, Gaithersbury, MD, USA, 261–269. https://doi.org/10.1109/ISSRE.2015.7381819

[3] Muhammad Ilyas Azeem, Fabio Palomba, Lin Shi, and Qing Wang. 2019. Machine Learning Techniques for Code Smell Detection: A Systematic Literature Review and Meta-Analysis. *Information and Software Technology* 108 (2019), 115–138. https://doi.org/10.1016/j.infsof.2018.12.009

[4] Antoine Barbez, Foutse Khomh, and Yann-Gaël Guéhéneuc. 2020. A Machine-Learning Based Ensemble Method for Anti-Patterns Detection. *Journal of Systems and Software* 161 (2020), 110486. https://doi.org/10.1016/j.jss.2019.110486

[5] Gabriele Bavota, Rocco Oliveto, Malcom Gethers, Denys Poshyvanyk, and Andrea De Lucia. 2014. Methodbook: Recommending Move Method Refactorings via Relational Topic Models. *IEEE Transactions on Software Engineering* 40, 7 (2014), 671–694. https://doi.org/10.1109/TSE.2013.60

[6] Leo Breiman. 1984. *Classification and Regression Trees*. Wadsworth, Belmont, CA, USA.

[7] Andrei Z Broder. 1997. On the Resemblance and Containment of Documents. In *Proceedings of Compression and Complexity of SEQUENCES (SEQUENCES '97)*. IEEE, Salerno, Italy, 21–29. https://doi.org/10.1109/SEQUEN.1997.666900

[8] Flavio Chierichetti, Ravi Kumar, Sandeep Pandey, and Sergei Vassilvitskii. 2010. Finding the Jaccard Median. In *Proceedings of the 21st Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '10)*. SIAM, Austin, TX, USA, 293–311. https://doi.org/10.1137/1.9781611973075.25.

[9] Di Cui, Siqi Wang, Yong Luo, Xingyu Li, Jie Dai, Lu Wang, and Qingshan Li. 2022. RMove: Recommending Move Method Refactoring Opportunities using Structural and Semantic Representations of Code. In *Proceedings of the 38th IEEE International Conference on Software Maintenance and Evolution (ICSME '22)*. IEEE, Limassol, Cyprus, 281–292. https://doi.org/10.1109/ICSME55016.2022.00033

[10] Marco D'Ambros, Alberto Bacchelli, and Michele Lanza. 2010. On the Impact of Design Flaws on Software Defects. In *Proceedings of the 10th International Conference on Quality Software (QSIC '10)*. IEEE, Zhangjiajie, China, 23–31. https://doi.org/10.1109/QSIC.2010.58

[11] Kris De Volder. 1998. *Type-Oriented Logic Meta Programming*. Ph. D. Dissertation. Vrije Universiteit Brussel, Programming Technology Laboratory.

[12] Chris Dennis. 2023. *Quartz*. https://github.com/quartz-scheduler/quartz/blob/ff348f62ece41275555b45dba5a4073c910fbeab/quartz-core/src/main/java/org/quartz/impl/StdSchedulerFactory.java/#L1404.

[13] Dario Di Nucci, Fabio Palomba, Damian A Tamburri, Alexander Serebrenik, and Andrea De Lucia. 2018. Detecting Code Smells Using Machine Learning Techniques: Are We There Yet?. In *Proceedings of the 25th International Conference on Software Analysis, Evolution and Reengineering (SANER '18)*. IEEE, Campobasso, Italy, 612–621. https://doi.org/10.1109/SANER.2018.8330266

[14] Danny Dig, Can Comertoglu, Darko Marinov, and Ralph Johnson. 2006. Automated Detection of Refactorings in Evolving Components. In *Proceedings of the 20th European Conference on Object-Oriented Programming (ECOOP '06)*. Springer, Nantes, France, 404–428. https://doi.org/10.1007/11785477_24

[15] Eduardo Fernandes, Johnatan Oliveira, Gustavo Vale, Thanis Paiva, and Eduardo Figueiredo. 2016. A Review-Based Comparative Study of Bad Smell Detection Tools. In *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering (EASE '16)*. ACM, Limerick, Ireland, 1–12. https://doi.org/10.1145/2915970.2915984

[16] Joseph L Fleiss. 1971. Measuring Nominal Scale Agreement Among Many Raters. *Psychological Bulletin* 76, 5 (1971), 378–382. https://doi.org/10.1037/h0031619.

[17] Marios Fokaefs, Nikolaos Tsantalis, and Alexander Chatzigeorgiou. 2007. JDeodorant: Identification and Removal of Feature Envy Bad Smells. In *Proceedings of the 23rd IEEE International Conference on Software Maintenance (ICSM '07)*. IEEE, Paris, France, 519–520. https://doi.org/10.1109/ICSM.2007.4362679

[18] Francesca Arcelli Fontana, Pietro Braione, and Marco Zanoni. 2012. Automatic Detection of Bad Smells in Code: An Experimental Assessment. *Journal of Object Technology* 11, 2 (2012), 5:1–38. https://doi.org/10.5381/jot.2012.11.2.a5

[19] 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. https://doi.org/10.1007/s10664-015-9378-4

[20] Francesca Arcelli Fontana, Elia Mariani, Andrea Mornioli, Raul Sormani, and Alberto Tonello. 2011. An Experience Report on Using Code Smells Detection Tools. In *Proceedings of the 4th IEEE International Conference on Software Testing, Verification and Validation Workshop on Refactoring and Testing Analysis (ICSTW '11)*. IEEE, Berlin, Germany, 450–457. https://doi.org/10.1109/ICSTW.2011.12

[21] Francesca Arcelli Fontana and Marco Zanoni. 2017. Code Smell Severity Classification Using Machine Learning Techniques. *Knowledge Based Systems* 128 (2017), 43–58. https://doi.org/10.1016/j.knosys.2017.04.014

[22] Francesca Arcelli Fontana, Marco Zanoni, Alessandro Marino, and Mika Mäntylä. 2013. Code Smell Detection: Towards a Machine Learning-Based Approach. In *Proceedings of the 2013 IEEE International Conference on Software Maintenance (ICSM '13)*. IEEE, Eindhoven, Netherlands, 396–399. https://doi.org/10.1109/ICSM.2013.56

[23] Martin Fowler. 1999. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, USA.

[24] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Pearson Deutschland GmbH, Munich, Germany.

[25] Adnane Ghannem, Ghizlane El Boussaidi, and Marouane Kessentini. 2016. On the Use of Design Defect Examples to Detect Model Refactoring Opportunities. *Software Quality Journal* 24, 4 (2016), 947–965. https://doi.org/10.1007/s11219-015-9271-9

[26] Gary Gregory. 2023. *Commons-CSV*. https://github.com/apache/commons-csv/blob/db374369aeeb1f3ace8efcbd7155fcff20354504/src/main/java/org/apache/commons/csv/Lexer.java#L128.

[27] Mouna Hadj-Kacem and Nadia Bouassida. 2019. Deep Representation Learning for Code Smells Detection using Variational Auto-Encoder. In *Proceedings of the 2019 International Joint Conference on Neural Networks (IJCNN '19)*. IEEE, Budapest, Hungary, 1–8. https://doi.org/10.1109/IJCNN.2019.8851854

[28] Eclipse JDT. 2023. *The Preconditions for Move Instance Method Refactorings*. https://github.com/eclipse-jdt/eclipse.jdt.ui/blob/master/org.eclipse.jdt.core.manipulation/core%20extension/org/eclipse/jdt/internal/corext/refactoring/structure/MoveInstanceMethodProcessor.java.

[29] Eclipse JDT. 2023. *The Preconditions for Move Static Method Refactorings*. https://github.com/eclipse-jdt/eclipse.jdt.ui/blob/master/org.eclipse.jdt.core.manipulation/core%20extension/org/eclipse/jdt/internal/corext/refactoring/structure/MoveStaticMembersProcessor.java.

[30] Thomas Junk. 1999. Confidence Level Computation for Combining Searches with Small Statistics. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment* 434, 2 (1999), 435–443. https://doi.org/10.1016/S0168-9002(99)00498-2

[31] René Just, Darioush Jalali, and Michael D Ernst. 2014. Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA '14)*. ACM, San Jose, CA, USA, 437–440. https://doi.org/10.1145/2610384.2628055

[32] Amandeep Kaur, Sushma Jain, and Shivani Goel. 2017. A Support Vector Machine Based Approach for Code Smell Detection. In *Proceedings of the 2017 International Conference on Machine Learning and Data Science (MLDS '17)*. IEEE, Noida, India, 9–14. https://doi.org/10.1109/MLDS.2017.8

[33] Taskin Kavzoglu. 2009. Increasing the Accuracy of Neural Network Classification Using Refined Training Data. *Environmental Modelling & Software* 24, 7 (2009), 850–858. https://doi.org/10.1016/j.envsoft.2008.11.012

[34] Keras. 2023. *Activation Functions*. https://github.com/keras-team/keras/blob/master/keras/activations.py.

[35] Miryung Kim, Matthew Gee, Alex Loh, and Napol Rachatasumrit. 2010. Ref-Finder: A Refactoring Reconstruction Tool Based on Logic Query Templates. In *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '10)*. ACM, Santa Fe, NM, USA, 371–372. https://doi.org/10.1145/1882291.1882353

[36] Jochen Kreimer. 2005. Adaptive Detection of Design Flaws. *Electronic Notes in Theoretical Computer Science* 141, 4 (2005), 117–136. https://doi.org/10.1016/j.entcs.2005.02.059

[37] Zarina Kurbatova, Ivan Veselov, Yaroslav Golubev, and Timofey Bryksin. 2020. Recommendation of Move Method Refactoring Using Path-Based Representation of Code. In *Proceedings of the 4th International Workshop on Refactoring (IWoR '20)*. ACM, Seoul, South Korea, 315–322. https://doi.org/10.1145/3387940.3392191

[38] Bo Liu. 2023. feTruth. https://github.com/lyoubo/feTruth.

[39] Bo Liu. 2023. Replication Package. https://doi.org/10.5281/zenodo.8267775.

[40] Hui Liu, Jiahao Jin, Zhifeng Xu, and Yifan Bu. 2023. *Pre-Trained Word2Vec Model*. https://doi.org/10.5281/zenodo.5749111.

[41] Hui Liu, Jiahao Jin, Zhifeng Xu, Yanzhen Zou, Yifan Bu, and Lu Zhang. 2021. Deep Learning Based Code Smell Detection. *IEEE Transactions on Software Engineering* 47, 9 (2021), 1811–1837. https://doi.org/10.1109/TSE.2019.2936376

[42] Hui Liu, Yuting Wu, Wenmei Liu, Qiurong Liu, and Chao Li. 2016. Domino Effect: Move More Methods Once a Method is Moved. In *Proceedings of the IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER '16)*. IEEE, Osaka, Japan, 1–12. https://doi.org/10.1109/SANER.2016.14

[43] Hui Liu, Zhifeng Xu, and Yanzhen Zou. 2018. Deep Learning Based Feature Envy Detection. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18)*. ACM, Montpellier, France, 385–396. https://doi.org/10.1145/3238147.3238166

[44] Xu-Ying Liu, Jianxin Wu, and Zhi Hua Zhou. 2009. Exploratory Undersampling for Class-Imbalance Learning. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)* 39, 2 (2009), 539–550. https://doi.org/10.1109/TSMCB.2008.2007853

[45] Bafandeh Bahareh Mayvan, Abbas Rasoolzadegan, and Javan Abbas Jafari. 2020. Bad Smell Detection Using Quality Metrics and Refactoring Opportunities. *Journal of Software: Evolution and Process* 32, 8 (2020), e2255. https://doi.org/10.1002/smr.2255

[46] Tom Mens and Tom Tourwé. 2004. A Survey of Software Refactoring. *IEEE Transactions on Software Engineering* 30, 2 (2004), 126–139. https://doi.org/10.1109/TSE.2004.1265817

[47] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient Estimation of Word Representations in Vector Space. In *Proceedings of the 1st International Conference on Learning Representations (ICLR '13)*. Scottsdale, AZ, USA, 1–12. https://doi.org/10.48550/arXiv.1301.3781

[48] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed Representations of Words and Phrases and Their Compositionality. In *Proceedings of the 27th Annual Conference on Neural Information Processing Systems (NIPS '13)*. Curran Associates, Inc., Lake Tahoe, NV, USA, 3111–3119.

[49] Haris Mumtaz, Mohammad Alshayeb, Sajjad Mahmood, and Mahmood Niazi. 2018. An Empirical Study to Improve Software Security Through the Application of Code Refactoring. *Information and Software Technology* 96 (2018), 112–125. https://doi.org/10.1016/j.infsof.2017.11.010

[50] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. 2013. Detecting Bad Smells in Source Code Using Change History Information. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering (ASE '13)*. IEEE, Silicon Valley, CA, USA, 268–278. https://doi.org/10.1109/ASE.2013.6693086

[51] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Denys Poshyvanyk, and Andrea De Lucia. 2015. Mining Version Histories for Detecting Code Smells. *IEEE Transactions on Software Engineering* 41, 5 (2015), 462–489. https://doi.org/10.1109/TSE.2014.2372760

[52] Fabio Palomba, Annibale Panichella, Andrea De Lucia, Rocco Oliveto, and Andy Zaidman. 2016. A Textual-Based Technique for Smell Detection. In *Proceedings of the 24th IEEE International Conference on Program Comprehension (ICPC '16)*. IEEE, Austin, TX, USA, 1–10. https://doi.org/10.1109/ICPC.2016.7503704

[53] Fabiano Pecorelli, Fabio Palomba, Dario Di Nucci, and Andrea De Lucia. 2019. Comparing Heuristic and Machine Learning Approaches for Metric-Based Code Smell Detection. In *Proceedings of the 27th International Conference on Program Comprehension (ICPC '19)*. IEEE, Montreal, QC, Canada, 93–104. https://doi.org/10.1109/ICPC.2019.00023

[54] Hao Peng, Lili Mou, Ge Li, Yuxuan Liu, Lu Zhang, and Zhi Jin. 2015. Building Program Vector Representations for Deep Learning. In *Proceedings of the 8th International Conference on Knowledge Science, Engineering and Management (KSEM '15)*. Springer, Chongqing, China, 547–553. https://doi.org/10.1007/978-3-319-25159-2_49

[55] Kyle Prete, Napol Rachatasumrit, Nikita Sudan, and Miryung Kim. 2010. Template-Based Reconstruction of Complex Refactorings. In *Proceedings of the 26th IEEE International Conference on Software Maintenance (ICSM '10)*. IEEE, Timisoara, Romania, 1–10. https://doi.org/10.1109/ICSM.2010.5609577

[56] Vitor Sales, Ricardo Terra, Luis Fernando Miranda, and Marco Tulio Valente. 2013. Recommending Move Method Refactorings Using Dependency Sets. In *Proceedings of the 20th Working Conference on Reverse Engineering (WCRE '13)*. IEEE, Koblenz, Germany, 232–241. https://doi.org/10.1109/WCRE.2013.6671298

[57] Gerard Salton and Michael J McGill. 1984. *Introduction to Modern Information Retrieval*. McGraw-Hill, New York, NY, USA.

[58] Tushar Sharma, Vasiliki Efstathiou, Panos Louridas, and Diomidis Spinellis. 2021. Code Smell Detection by Deep Direct-Learning and Transfer-Learning. *Journal of Systems and Software* 176 (2021), 110936. https://doi.org/10.1016/j.jss.2021.110936

[59] Danilo Silva, João Silva, Gustavo Jansen De Souza Santos, Ricardo Terra, and Marco Tulio O Valente. 2021. RefDiff 2.0: A Multi-Language Refactoring Detection Tool. *IEEE Transactions on Software Engineering* 47, 12 (2021), 2786–2802. https://doi.org/10.1109/TSE.2020.2968072

[60] Danilo Silva and Marco Tulio Valente. 2017. RefDiff: Detecting Refactorings in Version Histories. In *Proceedings of the 14th International Conference on Mining Software Repositories (MSR '17)*. IEEE, Buenos Aires, Argentina, 269–279. https://doi.org/10.1109/MSR.2017.14

[61] Frank Simon, Frank Steinbruckner, and Claus Lewerentz. 2001. Metrics Based Refactoring. In *Proceedings of the 5th European Conference on Software Maintenance and Reengineering (CSMR '01)*. IEEE, Lisbon, Portugal, 30–38. https://doi.org/10.1109/CSMR.2001.914965

[62] Dag IK Sjøberg, Bente Anda, and Audris Mockus. 2012. Questioning Software Maintenance Metrics: A Comparative Case Study. In *Proceedings of the 2012 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM '12)*. ACM, Lund, Sweden, 107–110. https://doi.org/10.1145/2372251.2372269

[63] Yan-Yan Song and Ying Lu. 2015. Decision Tree Methods: Applications for Classification and Prediction. *Shanghai Arch Psychiatry* 27, 2 (2015), 130–135. http://dx.doi.org/10.11919/j.issn.1002-0829.215044

[64] 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

[65] Martin Thompson. 2023. *Aeron*. https://github.com/real-logic/aeron/commit/305c060d424fdea13b1b7f2979d545ac7da5f7a5#diff-07eb841a7b48861016496075f986f4c6b330e72f8246e3270be31d9305cbd2bcR104.

[66] 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

[67] Nikolaos Tsantalis, Ameya Ketkar, and Danny Dig. 2022. RefactoringMiner 2.0. *IEEE Transactions on Software Engineering* 48, 3 (2022), 930–950. https://doi.org/10.1109/TSE.2020.3007722

[68] Nikolaos Tsantalis, Matin Mansouri, Laleh Eshkevari, Davood Mazinanian, and Danny Dig. 2018. Accurate and Efficient Refactoring Detection in Commit History. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. ACM, Gothenburg, Sweden, 483–494. https://doi.org/10.1145/3180155.3180206

[69] Aiko Yamashita and Leon Moonen. 2013. Do Developers Care about Code Smells? An Exploratory Survey. In *Proceedings of the 20th Working Conference on Reverse Engineering (WCRE '13)*. IEEE, Koblenz, Germany, 242–251. https://doi.org/10.1109/WCRE.2013.6671299

[70] Nico Zazworka, Michele A Shaw, Forrest Shull, and Carolyn Seaman. 2011. Investigating the Impact of Design Debt on Software Quality. In *Proceedings of the 2nd Workshop on Managing Technical Debt (MTD '11)*. ACM, Honolulu, HI, USA, 17–23. https://doi.org/10.1145/1985362.1985366

Received 2023-02-02; accepted 2023-07-27