

RMove: Recommending Move Method Refactoring Opportunities using Structural and Semantic Representations of Code

Di Cui, Siqi Wang, Yong Luo, Xingyu Li, Jie Dai, Lu Wang, Qingshan Li*
School of Computer Science and Technology, Xidian University, Xi'an 710049, China
cuidi@xidian.edu.cn; qshli@mail.xidian.edu.cn

Abstract—Incorrect placement of methods within classes is a typical code smell called Feature Envy, which causes additional maintenance and cost during evolution. To remove this design flaw, several Move Method refactoring tools have been proposed. To the best of our knowledge, state-of-the-art related techniques can be broadly divided into two categories: the first line is non-machine-learning-based approaches built on software measurement, while the selection and thresholds of software metrics heavily rely on expert knowledge. The second line is machine learning-based approaches, which suggest Move Method refactoring by learning to extract features from code information. However, most approaches in this line treat different forms of code information identically, disregarding their significant variation on data analysis. In this paper, we propose an approach to recommend Move Method refactoring named RMove by automatically learning structural and semantic representation from code fragment respectively. We concatenate these representations together and further train the machine learning classifiers to guide the movement of method to suitable classes. We evaluate our approach on two publicly available datasets. The results show that our approach outperforms three state-of-the-art refactoring tools including PathMove, JDeodorant, and JMove in effectiveness and usefulness. The results also unveil useful findings and provide new insights that benefit other types of feature envy refactoring techniques.

I. INTRODUCTION

For a long time, the significance of architectural design decisions has been acknowledged in the software research and industry community. The placement of methods within classes in an object-oriented system is a critical criterion for software architecture maintenance. During the software evolution, however, developers may inadvertently and unintentionally implement methods in inappropriate classes, resulting in a typical code smell: Feature Envy [1]. Previous studies [2, 3] also indicated that the Feature Envy is one of the most recurring code smells, negatively and seriously affecting the software system's maintainability.

To tackle this design flaw, several automatic Move Method refactoring approaches have been proposed, which moves the inappropriate method from its current class to its enviable class. This code transformation eliminates the Feature Envy by improving the internal code structure without changing its external behaviours. Most of these approaches can be more broadly divided into two categories: The first line is metric-based approaches, built on software measurement, such as

cohesion and coupling. Although these approaches can intuitively characterize Move Method refactoring from the structural and semantic perspective, the selection and thresholds of software metrics heavily rely on expert knowledge. The second line is machine learning-based approaches, which suggest Move Method refactoring by learning to extract features from the source code. However, in most cases, these approaches treat different forms of code information identically, disregarding their significant variation in data analysis. In reality, various forms of code information, such as structural information and semantic information, require drastically different machine learning algorithms for extracting features.

In this paper, we combined the comprehensive analysis of metric-based approaches and the automatic feature extraction of machine learning-based approach, and proposed an approach to recommend Move Method refactoring named RMove by learning structural and semantic representation of code fragment separately.

To capture the structural representation of code, we are motivated by the work of Qu et al. [4]. Their results demonstrated that the graph embedding technique: node2vec [5] can effectively characterize the topology of code structure and encode them into low-dimensional vector space as the structural representation of code. Their results presented that these extracted representations are proved to be practical in predicting bugs. Thus, in our method refactoring recommendation task, followed by the work of Qu et al. [4], we collect method dependency network as structural information. We further investigate 7 graph embedding techniques to capture structural representations of code based on collected data and make a systematic comparison.

To capture the semantic representation of code, we are motivated by the work of Alon et al. [6, 7]. Different from the previous techniques capturing program semantics from identifiers and comments using bag-of-words model [8–10], they use code embedding techniques learn continuous distributed vectors from AST paths using graph neural network as semantic representation, mapping semantically similar code snippets to close vectors. Their results also demonstrated that these extracted representations are proved to be useful in predicting method names. Therefore, in our method refactoring recommendation task, followed by the work of Alon et al. [6, 7], we collect AST path as semantic information. We

*Corresponding author.

employ 2 code embedding techniques to capture semantic representations of code based on collected data and further investigate their impact on the recommendation’s performance.

The procedure of our approach: RMove is demonstrated as follows: We first extract method structural and semantic information from the dataset. Next, we create the structural and semantic representation of collected data. We further normalized and fuse them together as the hybrid representation. Finally, based on these hybrid representations, we train machine learning classifier to suggest moving a target method to a more structurally and semantically similar class. We evaluate our approach using two publicly available datasets including a synthetic dataset of injected instances [11] and a real-world dataset of instances annotated by experts [12]. In terms of accuracy and effectiveness, we make a systematic comparison of our approach and other state-of-the-art tools. The results suggest that our approach outperforms state-of-the-art tools such as PathMove [13], JDeodorant [14], and JMove [12]. The results also unveil useful findings and provide new insights that benefit other types of feature envy refactoring techniques like move field, move class and move package.

In summary, we make the following contributions:

- A new perspective to recommend Move Method refactoring opportunities by exploiting structural and semantic representations of code snippets.
- A systematic exploration of implementations of our approach based on combinations of 2 code embedding techniques, 7 graph embedding techniques, and 9 machine learning classifiers. The results suggest that Code2Vec+SDNE (CV+SN), Code2Seq+Line (CS+LN), and Code2Seq+SDNE (CS+SN) achieve the best results.
- A comprehensive evaluation of our approach on the publicly available dataset. Our approach demonstrates an increase of 14%-36% in precision, 19%-45% in recall, and 27%-44% in f1-measure compared to stat-of-the-art tools: PathMove, JDeodorant, and JMove.
- A benchmark to investigate the effectiveness of structural and semantic representation of code snippets on two widely used datasets. All data are publically available [15].

II. PRELIMINARY

In this section, we explain the terminologies used in our paper.

Move Method Refactoring Detection. The Move Method refactoring detection can be regarded as discovering a set of movable methods and target classes from the source code, which is defined as *MoveMethodSet*. Each item in *MoveMethodSet* can be further modeled as a set of three elements, which is formally defined as follows:

$$MoveMethodSet = \{(m_i, sc_i, tc_i) \mid i = 1, 2, \dots, k\} \quad (1)$$

where m_i represents the potentially movable method. sc_i represents the source class which m_i belongs to. tc_i represents the corresponding target class for m_i .

To diagnose whether a method is movable, state-of-the-art related techniques analyze its structural and semantic information from the code snippets, which are further illustrated as follows:

Method Semantic Information. For each method, we first parse involved the code snippets into the Abstract Syntax Tree: *AST*, and further iteratively extract the path between leaf nodes from the parsed Abstract Syntax Tree as Method Semantic Information, which is formally defined as *PathSet*:

$$PathSet = \{(n_i, n_j, path(n_i, n_j)) \mid n_i, n_j \in AST\} \quad (2)$$

where n_i and n_j represent a pair of leaf nodes in Abstract Syntax Tree: *AST*. $path(n_i, n_j)$ represents the path between n_i and n_j , composed up of a sequence of intermediate AST nodes, which is obtained by traversing through their lowest common ancestor. Fig. 1 illustrates related concepts, including a fragment of code snippet, its corresponding parsed AST, and a path between two leaf nodes highlighted in blue. Arrows in blue in Fig. 1 present the path between two AST leaf nodes: b and a , which is represented as $Path(b, a): \{b \uparrow BinaryExpression \uparrow ConditionalExpression \downarrow a\}$. Furthermore, all paths: *PathSet* are gathered to obtain Method Semantic Information.

Method Structural Information. Given a method, we first extract Method Dependency Graph (MDG) from the source code and further analyze the topological structure for this method to obtain Method Structural Information. Method Dependency Graph is defined as:

$$MDG = \{V, E\} \quad (3)$$

where each node $v \in V$ represents a method and the edge $e \in E$ represents the method call dependency relationships. For a pair of methods: v_i and v_j , $(v_i, v_j) \in E$ if and only if v_i has at least one method call dependency relationship with v_j . Fig. 2 illustrates related concepts, including a fragment of code snippet and its corresponding Method Dependency Graph (MDG). These dependency relations: $\{(m_1, m_2), (m_1, m_3), (m_2, m_3)\}$ are presented.

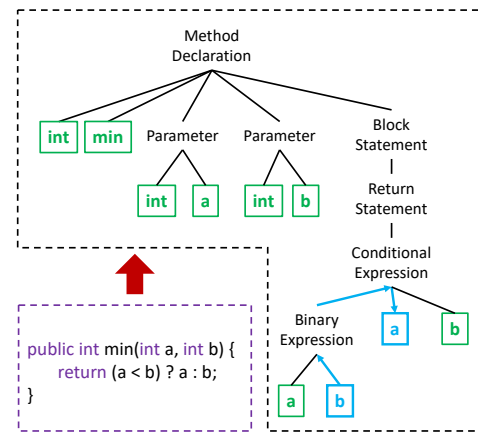


Fig. 1. A illustrated example of extracted method semantic information

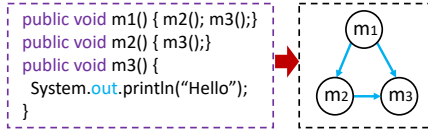


Fig. 2. A illustrated example of extracted method structural information

III. METHODOLOGY

Fig. 3 presents the overview of our proposed approach: RMove. We implement the automatic Move Method refactoring recommendation system by: 1) mining dataset to gather the Abstract Syntax Tree (AST) and the Method Dependency Graph (MDG), 2) generating hybrid representations with code embedding and graph embedding, and 3) training classification model using machine learning techniques and deep learning techniques. At last, we suggest Move Method refactoring according to the classification result.

A. Data Collection

We start with extracting all pairs of movable methods and target classes from the dataset, and then collect method structural and semantic information as follows:

Method Semantic Information Collection. We use SRCML [16], a state-of-the-art source code parser, to retrieve the source code of dataset’s collected movable methods and target classes. SRCML is a lightweight, scalable, multi-language parsing tool for converting source code into the XML format, which supports code exploration, analysis, and manipulation. We then mine AST paths: $PathSet$ with ASTMINER [17], a state-of-the-art static analysis tool, which is applied to the retrieved source code, including movable methods and target classes. ASTMINER is an open-source library for extracting AST paths from the source code. ASTMINER is a efficient, flexible, and extensible tool to support code analysis in various programming languages. We gather all extracted paths: $PathSet$ to obtain Method Semantic Information.

Method Structural Information Collection. we employ DEPENDS [18], a state-of-the-art static analysis tool, to extract dependencies among methods: MDG . DEPENDS is a dependency extraction tool for the source code that aims to infer dependency relationships between source code entities, such as files and methods, from various programming languages. DEPENDS also provides extensible interfaces to assist with downstream tasks such as architectural analysis and program comprehension. We gather all the Method Dependency Graph: MDG for each involved project in dataset as Method Structural Information.

B. Representation Generation

For collected method structural and semantic information, we use code embedding techniques and graph embedding techniques to generate its corresponding structural and semantic representations respectively, and further fuse them to generate hybrid representations, which are illustrated as follows:

Code Embedding Generation. Given each method: m_i and gathered AST path set: $PathSet$, the code embedding: CE is a mapping formally defined as:

$$CE = \{(m_i, cebd_i) \mid path(m_i) \in PathSet \wedge cebd_i \in R^d\} \quad (4)$$

where $path(m_i)$ represents the involved AST paths for m_i and $cebd_i$ represents the code embedding results for m_i . In this paper, we explore two state-of-the-art code embedding techniques including Code2Vec and Code2Seq. The basic ideas of these techniques are illustrated as follows:

Code2Vec [6] is a neural network that automatically generate vectors from code snippets. For AST paths of each method, Code2Vec learns numeric vectors for involved leaf nodes and intermediate nodes respectively. Code2Vec then concatenates these vectors together as a combined context vector. To generate the embedding result for code snippets, based on the attention mechanism, Code2Vec further calculates the weighted average of all combined context vectors by assigning more weights to AST paths with more significant semantics. The obtained fixed-length vectors for code snippets, referred as the code embedding result, can be further used in the downstream tasks.

Code2Seq [19] is also a neural network that produces sequences from code snippets. The model tasks the AST path set: $PathSet$ as input and generates distinct embedding results for tokens and paths in the AST path set before combining them into a single vector. To build embedding for tokens, Code2Seq first splits tokens into a sequence of subtokens according to Camel and Snake naming conventions, then transforms subtokens into a sequence of vectors with the embedding matrix: E^{token} , and finally combine these vectors to the token embedding. To build embedding for paths, code2seq embeds AST node types into numerical vectors with another embedding matrix E^{node} and thus produce a sequence of vectors. The obtained sequences then processed through an LSTM with the path’s embedding determined by the LSTM’s last state. The Code2Seq’s final step is to combine the generated vectors into the embedding result. To support this step, Code2Seq concatenates these vectors via a one-layer fully-connected neural network.

Implementations and hyper-parameter settings of code embedding techniques are introduced in Section IV-A, referred to previous papers [6, 19]. Table I presents the code embedding results of the method: validateLiteralPresence in the open source project: PMD [20]. We observed that these code embedding results are vastly different.

Graph Embedding Generation. For a Method Dependency Graph $MDG = \{V, E\}$, the graph embedding: GE is a mapping formally defined as:

$$GE = \{(m_i, gebd_i) \mid m_i \in V \wedge gebd_i \in R^d \wedge d \ll |V|\} \quad (5)$$

where m_i represents the node in Method Dependency Graph: MDG and $gebd_i$ represents the graph embedding results for m_i . In this paper, we explore seven state-of-the-art graph embedding techniques including DeepWalk, Node2Vec,

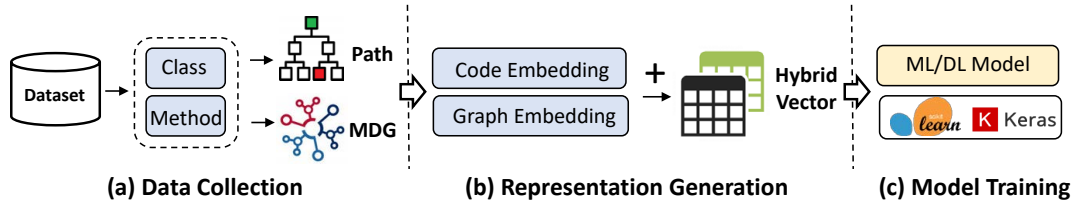


Fig. 3. Overview of the proposed approach RMove including data collection, representation generation, and model training.

TABLE I: The embedding results of the method: validateLiteralPresence in the open source project: PMD

Embedding	feature ₁	feature ₂	feature ₃	...	feature _n
Code2Vec	0.98	0.16	0.98	...	0.87
Code2Seq	0.18	-0.14	0.19	...	-0.06
DeepWalk	-1.25	0.62	0.19	...	-0.003
GraRep	0.18	-0.14	0.19	...	-0.06
Line	-0.02	-0.25	0.43	...	-0.06
Node2Vec	-0.07	0.01	-0.31	...	-0.03
ProNE	-0.004	-0.31	0.38	...	0.01
SDNE	3.94	4.57	2.76	...	-0.008
Walklets	-3.16	-1.37	-0.44	...	-0.04

Walklets, GraRep, Line, ProNE, and SDNE. The basic ideas of these techniques are illustrated as follows:

DeepWalk [21] and **Node2Vec** [5] employ random walk to construct sample neighborhoods for nodes in graph based on a Skip-gram [22] Natural Language Processing (NLP) model. The goal of Skip-gram is to maximize the likelihood of words appearing in a sliding window co-occurring. During random walks, each path sampled from a graph correlates to a sentence from the corpus in NLP, where each node correlates to a word. For these paths, the Skip-gram then is applied to the maximization of the probabilities of having a node's neighborhood based on its embedding results, which implemented with Stochastic Gradient Descent (SGD) and backpropagation on single hidden-layer feedforward neural network [5]. In comparison to DeepWalk, Node2Vec uses a more flexible notion of the node's neighborhoods and a more efficient graph searching algorithm achieving the trade-off between Breadth-first Sampling (BFS) and Depth-first Sampling (DFS) [23].

Walklets [24] is another random walk-based graph embedding technique. In comparison to DeepWalk and Node2Vec, Walklets explicitly encode multi-scale relationships between nodes to produce the multi-scale representations for nodes. Walklets first samples short random walks to extract multi-scale relationships. Furthermore, for each random walk, Walklets skips over steps and constructs latent representations, capturing higher order relationships from the adjacency matrix.

GraRep [25] is a matrix factorization-based graph embedding technique. These techniques construct matrices from connections between nodes and factorize them to produce the embedding result. The following matrices are frequently investigated including node adjacency matrix, node transition probability matrix, Laplacian matrix, etc. GraRep focuses on factorizing the node proximity matrix. The time complexity of

GraRep is $O(|V|^3)$, having a scalability issue [23].

Line [26] calculates graph embedding results by specifying two functions, one for the first-order node proximity and the other for the second-order node proximity. The line then minimizes the combination of these functions. For each pair of nodes with the first-order proximity, Line defines two joint probability distributions, one for the adjacency matrix and the other for the embedding. The Kullback-Leibler (KL) divergence of these distributions is then minimized. The calculation of the second-order proximity follows a similar pattern [23].

ProNE [27] is a fast and scalable graph embedding technique that was recently introduced. ProNE includes two steps: the first is to effectively initialize graph embedding results by phrasing the problem as sparse matrix factorization, motivated by the long-tailed distribution of most graphs and their sparsity. The second stage is to propagate the initial embedding result using the higher-order Cheeger's inequality [28], aiming at capturing the graph's localized clustering information. The experimental results [27] reveal that ProNE is 10 to 400 times faster than DeepWalk, Node2Vec and Line. ProNE's performance in the multi-label node classification task also outperformed existing graph embedding techniques.

SDNE [29] employs deep auto-encoders to generate embedding results. The objective of an auto-encoder is to reduce the reconstruction error. Multiple extremely non-linear functions are included in both the encoder and the decoder. The encoder converts input data into the representation space, and the decoder converts representation space into the reconstruction space [30]. There are two elements to the model: unsupervised and supervised. The first contains an auto-encoder to reconstruct the node's neighborhood and generate its embedding. The second is built on Laplacian Eigenmaps [31] and imposes a penalty when similar nodes are mapped with wrong results in the embedding space [23].

Implementations and hyper-parameter settings of graph embedding techniques are introduced in Section IV-A, referred to two recent survey papers [23, 30]. Table I also presents the graph embedding results of the method: validateLiteralPresence in the open source project: PMD [20]. We observed that these graph embedding results are drastically different.

Representation Fusing. We first normalize code embedding and graph embedding results respectively. Next, for each method: m_i , we concatenate its normalized code embedding and graph embedding as the hybrid embedding: $hebd(m_i)$ with

a tuning parameter: α , which is formally defined as:

$$\begin{aligned} hebd(m_i) &= [\alpha \times cebd_i, (1 - \alpha) \times gebd_i] (m_i, cebd_i) \in NCE \\ &\quad \wedge (m_i, gebd_i) \in NGE \end{aligned} \quad (6)$$

where $cebd_i$ represents the corresponding code embedding for m_i in the normalized embedding set: NCE and $gebd_i$ represents the corresponding code embedding results for m_i in the normalized embedding set: NGE . In our paper, we set the parameter α as 0.5, which means the hybrid embedding is split evenly between the code and graph embedding.

For each class: C_j , its corresponding hybrid embedding is calculated as an element-wise average of hybrid embedding results of contained methods:

$$hebd(C_j) = \frac{1}{|C_j|} \sum_{m_i \in C_j} hebd(m_i) \quad (7)$$

where $|C_j|$ represents the number of methods in class: C_j .

These fused embedding results for methods and classes are further employed in the model training process.

C. Model Training

For fused hybrid representations, we further generate training data and feed them with various classifiers to build the move method refactoring recommendation system, which is illustrated as follows:

Training Data Generation. We generate training data from a small set of detected move method detection results. Algorithm 1 presents the procedure of training data generation. The input of this algorithm is a set of detected move method results: *MoveMethodSet*. This algorithm inspects each item in *MoveMethodSet* iteratively. For each item, Line 4 retrieves movable method, source class and target class as m_i , sc_i , tc_i respectively. Line 6 concatenates hybrid embedding results of m_i and sc_i , and assigns with the false label as the negative sample. Line 8 concatenates hybrid embedding results of m_i and tc_i , and assigns with the true label as the positive sample. For example, for an instance of detected results concluding a movable method: a from the source class: A can be moved to the target class: B , we add a positive sample: “ m should be moved to B ” and a negative sample: “ m should not be moved to A ”. At the same time, if detected results also contain an instance that the method: m from the source class: A can be moved to a target class: C , we will additionally add a positive sample: “ m should be moved to C ” and a duplicate negative sample: “ m should be moved to A ” to balance the dataset.

Classifier Selection. Machine learning and deep learning models are frequently investigated for software data classification, which acquire classification knowledge through intensive training. In this paper, we employ 6 machine learning models and 3 deep learning models including Decision Tree (DT), Naive Bayes (NB), Support Vector Machine (SVM), Logistic Regression (LR), Random Forest (RF), Extreme Gradient Boosting (XGB), Convolutional Neural Network (CNN), Long

Algorithm 1 AutoTrainingDataGeneration(*MoveMethodSet*)

```

1: TrainData  $\leftarrow \emptyset$  % initialization
2: for item in MoveMethodSet do
3:   % getting method, source class and target class respectively
4:    $m_i, sc_i, tc_i \leftarrow item[0], item[1], item[2]$ 
5:   % adding negative samples
6:   TrainData.add(concat(hebd( $m_i$ ), hebd( $sc_i$ )), False)

7:   % adding positive samples
8:   TrainData.add(concat(hebd( $m_i$ ), hebd( $tc_i$ )), True)
9: end for
```

Short Term Memory Recurrent Neural Network (LSTM), and Gated Recurrent Units Recurrent Neural Network (GRU).

Decision Tree is a flowchart-like structural classifier frequently employed in fast data analysis tasks [32]. Naive Bayes is a classifier according to Bayes’ Theorem with the independent assumption of predictors [33]. Support Vector Machine is a supervised machine learning technique supporting both classification or regression tasks [34]. Logistic regression is a classifier that estimates probabilities between the categorical dependent variable and independent variables with the logistic or sigmoid function [35]. Random Forest [36] and Extreme Gradient Boosting [37] are ensemble models. Deep learning models, inspired by the human brain structure, have lately acquired enough attention in data classification. We employ widely used models in this field including Convolutional Neural Network (CNN) [38] and two variants of Recurrent Neural Network (RNN) [39]: RNN-LSTM, and RNN-GRU.

IV. EVALUATION

We design the evaluation to answer the following three research questions.

- RQ1: Which embeddings are the most effective in recommending move method refactoring?** The answer to this question would help us better understand the impact of different embedding combinations on RMove’s performance.
- RQ2: How accurate is RMove in recommending move method refactoring?** The answer to this question would demonstrate the RMove’s performance compared to state-of-the-art tools.
- RQ3: How useful is RMove in recommending move method refactoring?** The answer to this question would shed light on the RMove’s effectiveness in practice.

TABLE II: Synthetic Dataset’s Information

Subjects	#Version	#LOC	#Classes	#Methods	#MMethods
PMD	6.13.0	119,430	1,147	8,637	127
Cayenne	4.2	275,450	1,499	12,164	93
Pinpoint	1.9.0	290,974	2,551	17,024	105
Jenkins	1.51	155,667	768	6,292	38
Drools	7.22.0	680,234	2,758	27,793	256

TABLE III: Real-world Dataset’s Information

Subjects	#Version	#LOC	#Classes	#Methods	#MMethods
Weka	3.6.9	257,897	908	16,034	31
Ant	1.8.2	103,402	760	8,586	25
FreeCol	0.10.3	93,605	535	6,616	17
JMeter	2.5.1	81,222	682	7,392	25
FreeMind	0.9.0	53,782	368	4,074	12
JTOpen	7.8	340,752	1,450	22,143	39
DrJava	r5387	88,631	361	4,675	18
Maven	3.0.5	71,065	154	1,568	24

A. Experiment Setup

We illustrate the used dataset, evaluation, and experiment settings as follows:

Datasets. We evaluate RMove using two publicly available datasets: a synthetic dataset [11] and a real-world dataset [12].

Synthetic dataset. We use the synthetic dataset to evaluate the impact of various embedding techniques on RMove’s performance. The synthetic dataset contains five open-source projects with high quality including PMD [20], Cayenne [40], Pinpoint [41], Jenkins [42], and Drools [43]. The basic information of these subjects is presented in Table II, including the number of lines of source code (#LOC), number of classes (#Classes), number of methods (#Methods), and number of movable methods (#MMethods).

Real-world dataset. We use the real-world dataset to conduct a comparison of RMove and other state-of-the-art refactoring tools. Each instance in the real-world dataset is manually entered by experts and these data are also frequently investigated in previous work [12, 13]. The real-world dataset contains 8 open-source projects with high quality including Weak [44], Ant [45], FreeCol [46], JMeter [47], FreeMind [48], JTOpen [49], DrJava [50], and Maven [51]. The basic information of these subjects are also presented in Table III, including the number of lines of source code (#LOC), number of classes (#Classes), number of methods (#Methods), and number of movable methods (#MMethods).

Evaluation Metrics. Following the previous work [13], we use three widely-used evaluation metrics including precision, recall, and F1 score, defined as follows:

$$\text{Precision} = \frac{\# \text{ of correct refactorings}}{\# \text{ of recommended refactorings}} \quad (8)$$

$$\text{Recall} = \frac{\# \text{ of correct refactorings}}{\# \text{ of moved methods}} \quad (9)$$

$$\text{F1-Measure} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \quad (10)$$

We calculate precision as the ratio between the number of correct refactorings and the number of all recommended refactorings. We calculate recall as the ratio between the number of correct refactorings and the number of moved methods. We calculate F1-Measure as the harmonic mean of the precision and recall results.

Experiment Settings. We run the experiments on a 2.4GHz Intel Xeon-4210R server with 10 logical cores and 128GB of memory. We also follow most of default hyper-parameters of code embedding and graph embedding techniques in previous work [4, 6, 19], which is presented in Table IV.

In answering RQ1 (embedding evaluation), we train classifiers with various embedding techniques on the whole synthetic data and evaluate its effectiveness on RMove’s performance. We implement machine learning classifiers based on the python library: SCIKIT-LEARN [52] and deep learning classifiers based on the python library: KERAS [53]. We use the grid search strategies to automatically tune the hyper-parameters of classifiers [54]. We also repeat the 10-fold cross-validation 10 times (10×10) to reduce the bias caused by the randomness in experiments. The evaluation metrics are also calculated as the average value during these times.

In answering RQ2 (accuracy evaluation), we select the top 3 embedding combinations and related trained models according to the answer to RQ1. Furthermore, we evaluate the accuracy of the most effective models of these embedding combinations on the real-world dataset and compare these results to other state-of-the-art refactoring tools.

In answering RQ3 (usefulness evaluation), we conduct a human study by hiring 15 experienced software engineers to analyze recommended instances of move method refactoring for each refactoring tools in RQ2. All the participants are not the authors of this paper. Furthermore, we randomly select 5 instances from the detection result from each refactoring tool to reduce the complexity of analyzing the move method refactoring and thus maintain the concentration of participants.

B. Experiment Result

Embedding Evaluation (RQ1). To analyze the effectiveness of various embedding techniques, we conduct a systemic comparison of 14 combinations of 2 code embedding techniques and 7 graph embedding techniques. For each combination, we further investigate its performance with 9 classifiers on the synthetic dataset. Table V presents the performance of combinations of various embedding techniques on the synthetic data. For each column, we highlight the greatest precision, recall and f1-measure results with a grey background color and a + mark. For the row: “avg”, we also highlight the top 3 results of precision, recall, and F1-measure with a grey background color and a * mark. We further perform Kruskal-Wallis test and Dunnett’s test on the experimental results of 14 combinations, which suggests that there is a significant difference among these approaches. Accordingly, we label the top 3 embedding combinations with a grey background color. Fig. 4 further presents the average results of embedding combinations on various classifiers, where the vertical axis shows various embedding combinations, and the horizontal axis shows the precision, recall, and f1-measure respectively. For Fig. 4.(a)-(c), we label the top 3 results with a red color and corresponding embedding combinations with a * mark.

As presented in Table V and Fig. 4, we observed that 1) Combinations of Code2Vec+SDNE (CV+SN), code2Seq+Line (CS+LN), and Code2Seq+SDNE (CS+SN) outperform other embedding combinations. Code2Seq+SDNE (CS+SN) achieves the greatest results in precision, recall, and f1-measure. Code2Vec+SDNE (CV+SN) presents the greatest recall score and Code2Seq+Line (CS+LN) presents a relatively

TABLE IV: Hyper-parameter settings of code embedding and graph embedding techniques

Techniques	Hyper-parameter settings
Code2Vec	num_epochs=20, train_batch_size=1024, test_batch_size=1024, code_vector_size=128, path_embeddings_size =128 token_embeddings_size = 128, csv_buffer_size=100*1024*1024, default_embeddings_size = 128
Code2Seq	num_epochs=3000, test_batch_size=256, batch_size=256, shuffle_buffer_size=10000, max_path_length = 9 csv_buffer_size=100*1024*1024, max_contexts = 200, embeddings_size = 128, decoder_size = 128
DeepWalk	representation_size=128, clf_ratio=0.5, number_walks=10, walk_length=80, workers=8, window_size=10
GraRep	kstep=4
Line	representation_size=128, order=3, negative_ratio=5, clf_ratio=0.5
Node2Vec	p=0.25, q=0.25
ProNE	dimension=128, step=10, theta=0.5, mu=0.2
SDNE	alpha=1e-6, beta=5, nu1=1e-5, nu2=1e-4, batch_size=200, epoch=100
Walklets	dimensions=128, walk-number=5, walk_length=80, window_size=5, workers=4, min_count=1, p=1.0, q=1.0

TABLE V: The performance of combinations of various embedding techniques on the synthetic data. CV: Code2Vec, CS: Code2Seq, DW: DeepWalk, GR: GraRep, LN: Line, PN: ProNE, SN: SDNE, P: Precision, R: Recall, F1: F1-Measure.

Model	CV+DW			CV+GR			CV+LN			CV+NV			CV+PN			CV+SN			CV+WL		
	P	R	F1	P	R	F1	P	R	F1	P	R	F1	P	R	F1	P	R	F1	P	R	F1
DT	.78	.81 ⁺	.79	.82	.79	.80	.75	.81 ⁺	.78	.84	.80	.82 ⁺	.78	.81 ⁺	.80	.83	.85	.84 ⁺	.81 ⁺	.83 ⁺	.82 ⁺
NB	.55	.35	.43	.68	.48	.56	.71	.51	.60	.67	.46	.55	.66	.52	.58	.63	.86	.72	.58	.52	.55
SVM	.76	.74	.75	.73	.70	.71	.79	.78	.78	.80	.75	.78	.82	.76	.79	.70	.87	.78	.77	.79	.78
LR	.74	.74	.74	.71	.78	.74	.77	.77	.77	.77	.75	.76	.79	.80	.79	.75	.81	.77	.77	.75	.76
RF	.83	.80	.81	.81	.78	.79	.84 ⁺	.81 ⁺	.82 ⁺	.85 ⁺	.74	.79	.83	.76	.79	.85 ⁺	.80	.82	.80	.67	.73
XGB	.85 ⁺	.81 ⁺	.83 ⁺	.87 ⁺	.81 ⁺	.84 ⁺	.82	.74	.78	.80	.75	.78	.84 ⁺	.78	.81 ⁺	.85 ⁺	.81	.83	.79	.78	.78
CNN	.62	.74	.68	.61	.79	.69	.66	.68	.67	.54	.87 ⁺	.67	.66	.80	.72	.54	.92 ⁺	.68	.72	.72	.72
LSTM	.74	.73	.73	.65	.70	.67	.77	.78	.77	.76	.78	.77	.78	.79	.78	.76	.78	.77	.72	.76	.74
GRU	.73	.73	.74	.69	.47	.56	.77	.69	.73	.79	.67	.73	.80	.69	.74	.80	.73	.76	.78	.65	.71
Avg	.73	.72	.72	.73	.70	.71	.76	.73	.74	.76	.73	.74	.77	.74	.76	.74	.82 [*]	.78 [*]	.75	.72	.73
Model	CS+DW			CS+GR			CS+LN			CS+NV			CS+PN			CS+SN			CS+WL		
	P	R	F1	P	R	F1	P	R	F1	P	R	F1	P	R	F1	P	R	F1	P	R	F1
DT	.76	.75 ⁺	.75	.82	.71	.76	.84	.83	.83	.74	.82	.78	.81	.78	.80	.84	.89	.87	.78	.71 ⁺	.74 ⁺
NB	.58	.40	.47	.55	.33	.42	.66	.77	.71	.66	.41	.51	.69	.73	.71	.68	.94 ⁺	.79	.55	.59	.57
SVM	.73	.67	.70	.82	.77	.79	.82	.75	.78	.79	.73	.76	.80	.78	.79	.74	.83	.78	.75	.66	.70
LR	.68	.65	.66	.83	.77	.80	.84	.76	.80	.81	.75	.78	.83	.78	.80	.75	.75	.75	.68	.65	.66
RF	.85 ⁺	.69	.76 ⁺	.86 ⁺	.75	.78	.88	.81	.84	.87 ⁺	.81	.84 ⁺	.87 ⁺	.78	.82	.92 ⁺	.84	.88 ⁺	.84 ⁺	.66	.73
XGB	.81	.71	.76 ⁺	.83	.83 ⁺	.83 ⁺	.89 ⁺	.87	.88 ⁺	.86	.83 ⁺	.84 ⁺	.87 ⁺	.86 ⁺	.87 ⁺	.92 ⁺	.85	.88 ⁺	.78	.67	.72
CNN	.73	.67	.63	.65	.63	.62	.54	.96 ⁺	.70	.72	.67	.63	.73	.69	.65	.76	.68	.64	.77	.68	.63
LSTM	.68	.61	.64	.74	.63	.68	.83	.76	.79	.79	.72	.76	.79	.78	.78	.77	.81	.79	.63	.57	.60
GRU	.71	.54	.61	.71	.52	.60	.75	.63	.68	.78	.61	.68	.77	.68	.72	.78	.78	.78	.65	.57	.61
Avg	.72	.63	.66	.76	.66	.70	.78 [*]	.79 [*]	.78 [*]	.78 [*]	.71	.73	.79 [*]	.76	.77	.80 [*]	.82 [*]	.80 [*]	.71	.64	.66

TABLE VI: The accuracy of various refactoring tools on the real-world dataset. P: Precision, R: Recall, F1: F1-Measure.

Subjects	PathMove			JDeodorant			JMove			RMove-1			RMove-2			RMove-3		
	P	R	F1	P	R	F1	P	R	F1	P	R	F1	P	R	F1	P	R	F1
Weka	.224	.645	.332	.059	.548	.107	.108	.741	.189	.508	.959 ⁺	.664 ⁺	.514 ⁺	.917	.659	.485	.729	.583
Ant	.197	.56	.292	.171	.48	.252	.173	.84	.287	.491	.905 ⁺	.637 ⁺	.503 ⁺	.857	.634	.472	.69	.561
FreeCol	.048	.647	.089	.03	.294	.054	.074	.764	.135	.483	.879	.623	.513 ⁺	.906 ⁺	.655 ⁺	.506	.772	.611
JMeter	.264	.36	.305	.236	.52	.325	.275	.76	.404	.466	.923 ⁺	.62 ⁺	.469	.847	.604	.473 ⁺	.76	.583
FreeMind	.8 ⁺	.333	.47	.166	.583	.258	.148	.666	.242	.517	.964 ⁺	.673 ⁺	.524	.857	.65	.484	.764	.593
JTOpen	.416	.512	.459	.207	.447	.283	.208	.894 ⁺	.337	.484	.872	.623	.497	.854	.628	.521 ⁺	.812	.635 ⁺
DrJava	.428	.5	.461	.128	.555	.208	.128	.777	.22	.526 ⁺	.96 ⁺	.68 ⁺	.5	.825	.623	.518	.81	.632
Maven	.545 ⁺	.541	.543	.139	.25	.179	.104	.375	.163	.496	.853	.627	.505	.771	.61	.51	.876 ⁺	.645 ⁺
Avg	.365	.512	.369	.142	.46	.208	.152	.727	.247	.496	.914 [*]	.643 [*]	.503 [*]	.854	.633	.496	.777	.605

high precision and f1-measure score. A possible explanation is that SDNE comprehensively captures the method semantic information to improve the recall score whereas Code2Seq efficiently characterizes the method semantic information to increase the precision score. 2) Random Forest (RF) and Extreme Gradient Boosting (XGB) outperform other classifiers in most embedding combinations. Although Decision Tree (DT) performs effectively in specific embedding combinations, Random Forest (RF) and Extreme Gradient Boosting (XGB) present the best precision, recall and f1-measure results in most cases. This result demonstrates that deep learning classifiers do not perform as well as we expected in recommending move method refactorings. The reason might be that software data, unlike image and text, is more suitable for machine

learning classifiers.

Answer to RQ1: Code2Vec+SDNE (CV+SN), Code2Seq+Line (CS+LN), and Code2Seq+SDNE (CS+SN) are the most effective embedding combinations.

Accuracy Evaluation (RQ2). To evaluate the accuracy of RMove, we select three most effective embedding combinations: Code2Vec+SDNE (CV+SN), Code2Seq+Line (CS+LN), and Code2Seq+SDNE (CS+SN). Furthermore, we rank related trained models of these embedding combinations on the real-world dataset and label the most effective classifier for each combination. Therefore, we regard these combinations and related classifiers with the best performance as three variants of RMove: RMove-1, RMove-2, and RMove-3, which corresponds to Code2Vec+SDNE (CV+SN), Code2Seq+Line

(CS+LN), and Code2Seq+SDNE (CS+SN) respectively. Fig. 5 presents the performance of classifiers on selected three embedding combinations. Fig. 5.(a)-(c) presents the results of Code2Vec+SDNE (CV+SN), Code2Seq+Line (CS+LN), and Code2Seq+SDNE (CS+SN) respectively, while the most effective classifier was labeled with a * mark. Table VI presents the accuracy of various refactoring tools on the real-world dataset including PathMove [13], JDeodorant [14], JMove [12], RMove-1, RMove-2, and RMove-3. For each row/subject, we highlight the greatest precision, recall and f1-measure results with a grey background color and a + mark. Specifically, for the row: “avg”, we highlight the maximum average scores of precision, recall and f1-measure on studied subjects with a grey background color and a * mark. Accordingly, we label the most accurate refactoring tool with a grey background color.

As presented in Table VI and Fig. 5, we observed that 1) Naive Bayes outperforms other classifiers in all of selected embedding combinations including Code2Vec+SDNE (CV+SN), Code2Seq+Line (CS+LN), and Code2Seq+SDNE (CS+SN) on the real-world dataset. Naive Bayes has the greatest recall score of all classifiers, resulting in a higher f1-measure score. This result demonstrates that Naive Bayes outperforms Random Forest (RF) and Extreme Gradient Boosting (XGB) on the real-world dataset despite having the best performance on the synthetic dataset in RQ1. One possible reason might be that Naive Bayes is more prone to generalize to other drastically different datasets in comparison to other classifiers. 2). RMove demonstrates an increase of 14%-36% in precision, 19%-45% in recall, and 27%-44% in f1-measure compared to stat-of-the-art tools: PathMove [13], JDeodorant [14], and JMove [12] while the statical test results also show that RMove is significantly better than these three tools. Despite PathMove’s high precision on 2 subjects and JMove’s high recall on 1 subject, RMove outperforms other refactoring tools in precision, recall, and f1-measure on most subjects of the real-world dataset. Specially, RMove-1 presents the increase of recall scores while RMove-2 presents the improvement of precision scores. In real scenarios, software practitioners may choose the proper embedding combinations and classifiers as settings for recommending move method refactorings.

Answer to RQ2: RMove has an increase of 14%-36% in precision, 19%-45% in recall, and 27%-44% in f1-measure compared to stat-of-the-art refactoring tools.

Usefulness Evaluation (RQ3). To evaluate the usefulness of RMove, we conducted a user study with 15 participants to review detection results of 6 refactoring tools on the subject FreeMind in RQ2. The studied subject: FreeMind is suitable for understanding. All the participants are industrial engineers having at least 6 years of working experience, which are also not the authors of this paper. We offered the source code of FreeMind and 6 groups of detection results and each group has 5 instances, which have been sampled several times to limit the overlap of these samples. We blindly present each group of detection results in a random order to each participant

to ensure they do not know which tool was developed by us. After reviewing all the groups, participants are required to evaluate each group’s performance. We further asked the participants to complete a questionnaire about refactoring tools. For each tool, we ask participants the question “Would you apply the refactoring tool?” and provide them with 5 ranking options including “Definitely Not”, “No”, “Maybe”, “Yes”, and “Absolutely Yes”.

Table VII reports the results of the questionnaire of 15 participants. The first column presents the refactoring tool. The other columns present the answer of each participant for each refactoring tool. Fig. 6 presents the distribution of participants’ answers for each refactoring tool. We observed that 1) Generally, the majority of participants believe that RMove is more helpful compared with the results of state-of-the-art refactoring tools. 7-9 participants choose “Absolutely Yes” or “Yes” for RMove while 1-4 participants support other tools. 2) More participants believe that state-of-the-art refactoring tools are not very helpful. 6-8 participants choose “Definitely Not” or “No” for these refactoring tools. 3) More participants hesitate to use state-of-the-art refactoring tools. 5-6 participants choose “Maybe” for these refactoring tools. This result indicates that RMove is more likely than state-of-the-art refactoring tools to be embraced by software practitioners.

Answer to RQ3: RMove is more useful than other refactoring tools in recommendation of move method refactoring opportunities for most participants.

V. DISCUSSION

In this section, we discuss the limitation, threats to validity, and applications of our approach.

Threats. Our research has the following threats: The first threat comes from data used in our evaluation. We only evaluate our approach in a part of selected open-source projects. It is still unclear whether our approach will generalize to closed source industrial projects and open-source projects from other communities. Replicating our approach on more datasets is our ongoing work. The second threat comes from the data quality of our training data. The collected data may contain noise which may lead to the bias of model training. To limit this threat, we employ the popular open-source projects, which are maintained by active communities containing less noise. We further manually check each item of training data to improve its quality. The third threat comes from the feasible evaluation of our recommendation results. We employed 15 participants to manually check the preconditions of our move method recommendation results. In our future work, we will further leverage automatic precondition validation techniques to ensure the refactoring recommendation solutions are actually applicable. The fourth threat comes from the run-time evaluation of our approach. We only test the run-time of RMove on real-world dataset. On these small subjects, RMove takes a relatively long time to process data and train model. However, once these phrases are completed, RMove can return refactoring recommendation results rapidly. Table VIII presents the run-time performance of RMove on the real-world dataset. On

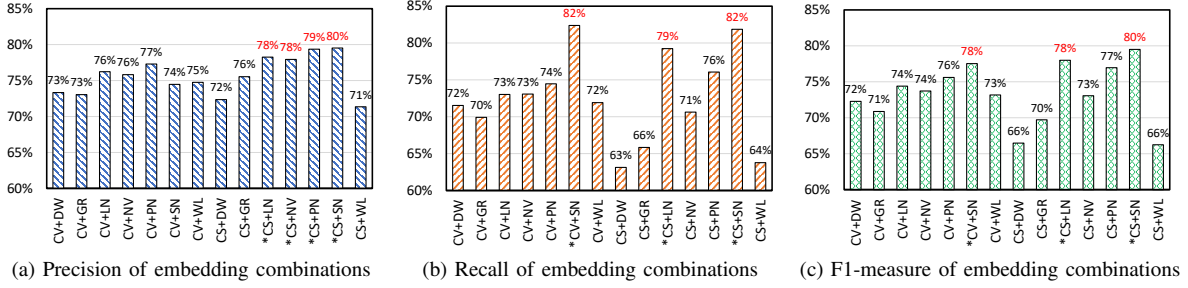


Fig. 4. The average precision, recall, and f1-measure of embedding combinations.

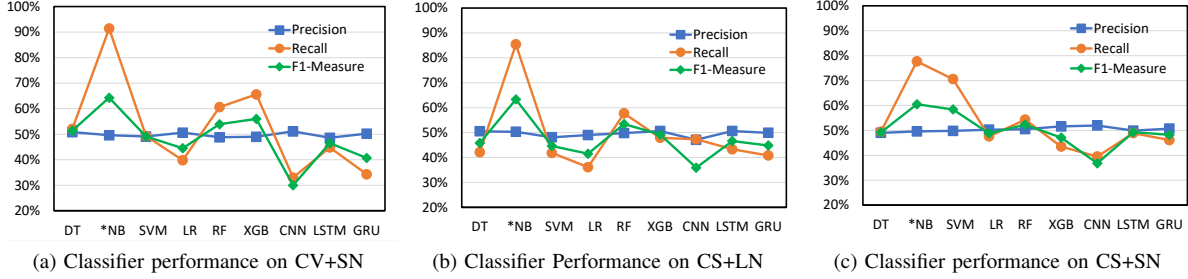


Fig. 5. Classifier performance on CV+SN, CS+LN, and CS+SN. CV: Code2Vec, CS: Code2Seq, SN: SDNE, LN: Line.

TABLE VII: Participants' answers to the question "Would you apply the refactoring tool?". AY: Absolutely Yes, Y: Yes, M: Maybe, N: No, DN: Definitely Not.

Tool	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14	P15
PathMove	AY	N	Y	M	N	M	M	Y	DN	M	N	N	DN	Y	M
JDeodorant	M	M	DN	N	N	M	N	N	DN	M	N	M	DN	M	Y
JMove	M	Y	M	N	M	N	M	DN	M	DN	N	AY	N	DN	N
RMove-1	AY	Y	N	AY	Y	AY	Y	N	Y	M	AY	N	M	Y	N
RMove-2	Y	M	N	AY	M	Y	Y	N	Y	DN	AY	N	M	AY	M
RMove-3	M	N	M	AY	Y	Y	AY	M	Y	M	Y	N	DN	Y	N

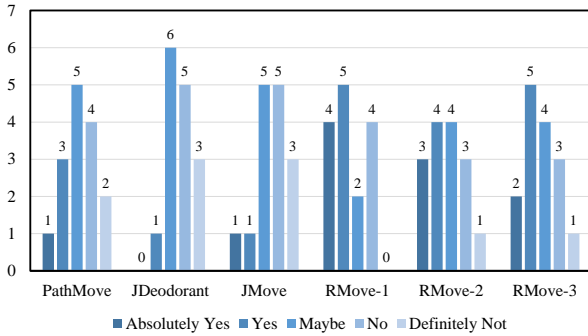


Fig. 6. The distribution of participants' answers for each refactoring tool.

average, RMove responds within 2 seconds. Analyzing the run-time performance of our approach on large scale subjects is our ongoing work.

Limitations. Our research has several limitations, which are illustrated as follows: First, machine learning techniques are significantly influenced by their hyper-parameters. To limit this threat, we employ grid search to explore suitable hyper-

parameters of classifiers. However, we follow most of the default hyper-parameters of embedding techniques used in previous study [4, 6, 19], which may cause sub-optimal results. In our future work, we will further explore the impact of hyper-parameters of embedding techniques including code embedding and network embedding. Second, we employ the implementation of embedding techniques in previous work [4, 6, 19], which may produce wrong results. To reduce this threat, we reported our found defects. If they are not fixed, we tried to fix them by ourselves. This can be further reduced by using more advanced tools. Third, we only focus on calling relationships between methods to generate structural representation. We did not consider fine-grained information of method calling behaviours as features in our approach, like various types of callees and calling frequencies. We believe these fine-grained features can be potentially useful and will further leverage them to improve our approach in future work.

Applications. Our research can be further extended in several directions, which are listed as follows: First, our approach uses the synthetic dataset to train the classification model and some results are promising. Despite its small size, the synthetic

TABLE VIII: The run-time performance of RMove.

Tool	Weka	Ant	FreeCol	JMeter	FreeMind	JTOpen	DrJava	Maven	Avg
Rmove-1	1.45s	1.86s	1.97s	1.46s	1.37s	1.52s	1.30s	1.47s	1.55s
Rmove-2	1.06s	1.44s	1.51s	1.06s	0.93s	1.08s	0.90s	1.07s	1.13s
Rmove-3	1.61s	1.96s	2.06s	1.53s	1.47s	1.62s	1.47s	1.55s	1.66s

dataset is of great quality. This implies that in refactoring recommendation tasks, data quality is more significant than data quantity. In future work, high-quality synthetic data could be leveraged to improve the performance of refactoring recommendation techniques. Second, our results indicate that various embedding combinations have drastically different effects on the refactoring recommendation results. Furthermore, the performance of classifiers also varies depending on the embedding combination. This opens up the possibility of using the ensemble technique to combine several variants of our approach implemented with various embedding combinations and classifiers. Designing the proper ensemble technique to improve the refactoring recommendation in future work is deserved. Third, our results demonstrate that improving the move method refactoring recommendation using structural and semantic representations of code snippets is feasible. It implies that the combination of structural and semantic representations could help with further additional feature envy refactoring techniques such as move attribute, move class and move package, which are all related to the inappropriate placement of code entities. Our future work will focus on the existing refactoring recommendation technique improving by exploiting both the structural and semantic representation of code snippets.

VI. RELATED WORK

Over the past decades, numerous tools have been designed to automatically suggest Move Method refactoring. In this section, we provide an overview of existing approaches and categorize them broadly into two groups:

Non-machine-learning-based approaches. The most representative work in this area is JDeodorant, which was introduced by Tsantalis et al. [14]. JDeodorant supports the detection of several types of code smells, including Feature Envy, Long Method, and God Class, and recommend appropriate refactoring suggestions to remove them. JDeodorant detects Feature Envy based on the following rule: a method should be moved when this method accesses more entities in other classes than entities in its class. To ensure the correctness of Move Method refactoring suggestions, JDeodorant also employs a set of preconditions and automatically verifies them. Terra et al. [12] introduced a Move Method refactoring tool: JMove based on the similarity between dependency sets. JMove considers dependencies such as method calls, field accesses, return types, etc. JMove detects Feature Envy according to the dependencies established by the method. The results show that JMove outperforms JDeodorant in accuracy and efficiency, especially for large methods. Liu et al. [55] introduced a Move Method refactoring tool: Domino based on the movement of other methods. Domino detects Feature Envy based on a heuristic that similar methods should be

moved together. When a method is moved, Domino explores possible methods and suggests to move. Ujihara et al. [56] introduced a Move Method refactoring tool: C-JRefRec based on static program analysis. C-JRefRec detects Feature Envy by computing the semantic similarity between the method and target class with TF-IDF vectors. Bavota et al. [57] introduced a Move Method refactoring tool: MethodBook using Relational Topic Model (RTM). MethodBook considers both method calls and textual information, including identifiers and comments, to support the detection of Feature Envy. These proposed approaches are based on software metrics, which require expert knowledge to define rules and thresholds, whereas our approach employs embedding techniques to learn features from code snippets automatically.

Machine learning-based approaches. Liu et al. [58] proposed a deep learning-based approach to detect several code smells including Feature Envy, Long Method, and Large Class. They first extract textual information such as identifiers and generate its representations with the word2vec technique [22]. Then, they further calculate the distance between representations and use Convolutional Neural Network (CNN) model to identify Feature Envy. Hadj-Kacem et al. [59] also proposed a deep learning-based approach to detect several code smells including Feature Envy, Long Method, and Blob. They parse the source code into Abstract Syntax Trees (AST) and generate its representation using the Variational Auto-Encoder (VAE) [60]. They further use the Linear Regress classifier to detect Feature Envy. Sharma et al. [61] systematically compare the detection of code smells including Complex Method, Magic Number, Empty Catch Block, and Multifaceted Abstraction with deep learning models. The results present that Recurrent Neural Network performs the best. Bryksin et al. [62] introduced a Move Method refactoring tool: ArchReload using clustering ensemble techniques. ArchReload combines the detection results of several heuristic-based approaches, such as ARI [63], HAC [64], and CCDA [65]. Barbez et al. [66] also introduced a Move Method refactoring tool with classifier ensemble techniques. They first collect software metrics and further employ these metrics to train machine learning classifiers and ensemble them. These approaches extract identifiers from code snippets and characterize features using word2vec techniques, while our approach uses graph embedding techniques and code embedding techniques to better capture the structure and semantic properties of code snippets.

VII. CONCLUSION

In this paper, we proposed an approach to recommend Move Method refactoring named RMove by automatically learning both structural and semantic representation from code snippets. We first extract method structural and semantic information from the dataset. Next, we create the structural and semantic

representation, and further concatenate these representations. Finally, we train the machine learning classifier to guide the movement of method to suitable class. The results show that our approach outperforms three state-of-the-art refactoring tools including PathMove, JDeodorant, and JMove.

ACKNOWLEDGEMENT

This work was supported by National Natural Science Foundation of China (61902288, 61972300, U21B2015), Strategic Priority Research Program of Chinese Academy of Science(XDC05040100), National Key Research and Development Program of China under Grant (2019YFB1406404), Fundamental Research Funds for the Central Universities (XJS220311).

REFERENCES

- [1] M. Fowler, *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018.
- [2] M. D'Ambros, A. Bacchelli, and M. Lanza, "On the impact of design flaws on software defects," in *2010 10th International Conference on Quality Software*. IEEE, 2010, pp. 23–31.
- [3] D. I. Sjøberg, A. Yamashita, B. C. Anda, A. Mockus, and T. Dybå, "Quantifying the effect of code smells on maintenance effort," *IEEE Transactions on Software Engineering*, vol. 39, no. 8, pp. 1144–1156, 2012.
- [4] Y. Qu and H. Yin, "Evaluating network embedding techniques' performances in software bug prediction," *Empirical Software Engineering*, vol. 26, no. 4, pp. 1–44, 2021.
- [5] A. Grover and J. Leskovec, "node2vec: Scalable feature learning for networks," in *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, 2016, pp. 855–864.
- [6] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning distributed representations of code," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–29, 2019.
- [7] U. Alon, S. Brody, O. Levy, and E. Yahav, "code2seq: Generating sequences from structured representations of code," in *International Conference on Learning Representations*, 2019. [Online]. Available: <https://openreview.net/forum?id=H1gKY09tX>
- [8] G. Bavota, B. Dit, R. Oliveto, M. Di Penta, D. Poshyvanyk, and A. De Lucia, "An empirical study on the developers' perception of software coupling," in *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 2013, pp. 692–701.
- [9] D. Cui, T. Liu, Y. Cai, Q. Zheng, Q. Feng, W. Jin, J. Guo, and Y. Qu, "Investigating the impact of multiple dependency structures on software defects," in *Proceedings of the 41st International Conference on Software Engineering*. IEEE Press, 2019, pp. 584–595.
- [10] D. Cui, L. Fan, S. Chen, Y. Cai, Q. Zheng, Y. Liu, and T. Liu, "Towards characterizing bug fixes through dependency-level changes in apache java open source projects," *Information Sciences*, vol. 65, no. 172101, pp. 1–172 101, 2022.
- [11] E. Novozhilov, I. Veselov, M. Pravilov, and T. Bryksin, "Evaluation of move method refactorings recommendation algorithms: are we doing it right?" in *2019 IEEE/ACM 3rd International Workshop on Refactoring (IWor)*. IEEE, 2019, pp. 23–26.
- [12] R. Terra, M. T. Valente, S. Miranda, and V. Sales, "Jmove: A novel heuristic and tool to detect move method refactoring opportunities," *Journal of Systems and Software*, vol. 138, pp. 19–36, 2018.
- [13] Z. Kurbatova, I. Veselov, Y. Golubev, and T. Bryksin, "Recommendation of move method refactoring using path-based representation of code," in *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, 2020, pp. 315–322.
- [14] N. Tsantalis and A. Chatzigeorgiou, "Identification of move method refactoring opportunities," *IEEE Transactions on Software Engineering*, vol. 35, no. 3, pp. 347–367, 2009.
- [15] (2022) Data link. [Online]. Available: <https://github.com/wangsiqidahaoren/Rmove-method>
- [16] (2022) SRCML. [Online]. Available: <https://www.srcml.org>
- [17] (2022) ASTMINER. [Online]. Available: <https://github.com/JetBrains-Research/astminer>
- [18] (2022) DEPENDS. [Online]. Available: <https://github.com/multilang-depends/depends>
- [19] U. Alon, S. Brody, O. Levy, and E. Yahav, "code2seq: Generating sequences from structured representations of code," *arXiv preprint arXiv:1808.01400*, 2018.
- [20] (2022) Pmd. [Online]. Available: <https://pmd.github.io/>
- [21] B. Perozzi, R. Al-Rfou, and S. Skiena, "Deepwalk: Online learning of social representations," in *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2014, pp. 701–710.
- [22] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *arXiv preprint arXiv:1301.3781*, 2013.
- [23] P. Goyal and E. Ferrara, "Graph embedding techniques, applications, and performance: A survey," *Knowledge-Based Systems*, vol. 151, pp. 78–94, 2018.
- [24] B. Perozzi, V. Kulkarni, H. Chen, and S. Skiena, "Don't walk, skip! online learning of multi-scale network embeddings," in *Proceedings of the 2017 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining 2017*, 2017, pp. 258–265.
- [25] S. Cao, W. Lu, and Q. Xu, "Grarep: Learning graph representations with global structural information," in *Proceedings of the 24th ACM international on conference on information and knowledge management*, 2015, pp. 891–900.
- [26] J. Tang, M. Qu, M. Wang, M. Zhang, J. Yan, and Q. Mei, "Line: Large-scale information network embedding," in *Proceedings of the 24th international conference on*

world wide web, 2015, pp. 1067–1077.

- [27] J. Zhang, Y. Dong, Y. Wang, J. Tang, and M. Ding, “Prone: Fast and scalable network representation learning,” in *IJCAI*, vol. 19, 2019, pp. 4278–4284.
- [28] J. R. Lee, S. O. Gharan, and L. Trevisan, “Multiway spectral partitioning and higher-order cheeger inequalities,” *Journal of the ACM (JACM)*, vol. 61, no. 6, pp. 1–30, 2014.
- [29] D. Wang, P. Cui, and W. Zhu, “Structural deep network embedding,” in *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, 2016, pp. 1225–1234.
- [30] H. Cai, V. W. Zheng, and K. C.-C. Chang, “A comprehensive survey of graph embedding: Problems, techniques, and applications,” *IEEE Transactions on Knowledge and Data Engineering*, vol. 30, no. 9, pp. 1616–1637, 2018.
- [31] M. Belkin and P. Niyogi, “Laplacian eigenmaps and spectral techniques for embedding and clustering,” in *Nips*, vol. 14, no. 14, 2001, pp. 585–591.
- [32] S. R. Safavian and D. Landgrebe, “A survey of decision tree classifier methodology,” *IEEE transactions on systems, man, and cybernetics*, vol. 21, no. 3, pp. 660–674, 1991.
- [33] I. Rish *et al.*, “An empirical study of the naive bayes classifier,” in *IJCAI 2001 workshop on empirical methods in artificial intelligence*, vol. 3, no. 22, 2001, pp. 41–46.
- [34] W. S. Noble, “What is a support vector machine?” *Nature biotechnology*, vol. 24, no. 12, pp. 1565–1567, 2006.
- [35] D. W. Hosmer Jr, S. Lemeshow, and R. X. Sturdivant, *Applied logistic regression*. John Wiley & Sons, 2013, vol. 398.
- [36] M. Pal, “Random forest classifier for remote sensing classification,” *International journal of remote sensing*, vol. 26, no. 1, pp. 217–222, 2005.
- [37] T. Chen, T. He, M. Benesty, V. Khotilovich, Y. Tang, H. Cho *et al.*, “Xgboost: extreme gradient boosting,” *R package version 0.4-2*, vol. 1, no. 4, pp. 1–4, 2015.
- [38] R. Girshick, “Fast r-cnn,” in *Proceedings of the IEEE international conference on computer vision*, 2015, pp. 1440–1448.
- [39] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, “Learning phrase representations using rnn encoder-decoder for statistical machine translation,” *arXiv preprint arXiv:1406.1078*, 2014.
- [40] (2022) Cayenne. [Online]. Available: <https://cayenne.apache.org>
- [41] (2022) Pinpoint. [Online]. Available: <https://pinpoint-apm.github.io/pinpoint>
- [42] (2022) Jenkins. [Online]. Available: <https://www.jenkins.io>
- [43] (2022) Drools. [Online]. Available: <https://drools.org>
- [44] (2022) Weka. [Online]. Available: <https://www.cs.waikato.ac.nz/ml/weka>
- [45] (2022) Ant. [Online]. Available: <https://ant.apache.org>
- [46] (2022) Freecol. [Online]. Available: <http://www.freecol.org>
- [47] (2022) Jmeter. [Online]. Available: <https://jmeter.apache.org>
- [48] (2022) Freemind. [Online]. Available: <http://freemind.sourceforge.net>
- [49] (2022) Jtopen. [Online]. Available: <http://jt400.sourceforge.net>
- [50] (2022) Drjava. [Online]. Available: <http://www.drjava.org>
- [51] (2022) Maven. [Online]. Available: <https://maven.apache.org>
- [52] (2022) SCIKIT-LEARN. [Online]. Available: <https://scikit-learn.org>
- [53] (2022) KERAS. [Online]. Available: <https://keras.io/>
- [54] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto, “The impact of automated parameter optimization on defect prediction models,” *IEEE Transactions on Software Engineering*, vol. 45, no. 7, pp. 683–711, 2018.
- [55] H. Liu, Y. Wu, W. Liu, Q. Liu, and C. Li, “Domino effect: Move more methods once a method is moved,” in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1. IEEE, 2016, pp. 1–12.
- [56] N. Ujihara, A. Ouni, T. Ishio, and K. Inoue, “c-jrefrec: Change-based identification of move method refactoring opportunities,” in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2017, pp. 482–486.
- [57] G. Bavota, R. Oliveto, M. Gethers, D. Poshyanyk, and A. De Lucia, “Methodbook: Recommending move method refactorings via relational topic models,” *IEEE Transactions on Software Engineering*, vol. 40, no. 7, pp. 671–694, 2013.
- [58] H. Liu, J. Jin, Z. Xu, Y. Bu, Y. Zou, and L. Zhang, “Deep learning based code smell detection,” *IEEE transactions on Software Engineering*, 2019.
- [59] M. Hadj-Kacem and N. Bouassida, “Deep representation learning for code smells detection using variational auto-encoder,” in *2019 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 2019, pp. 1–8.
- [60] F. Palomba, G. Bavota, M. Di Penta, F. Fasano, R. Oliveto, and A. De Lucia, “On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation,” *Empirical Software Engineering*, vol. 23, no. 3, pp. 1188–1221, 2018.
- [61] T. Sharma, V. Efstathiou, P. Louridas, and D. Spinellis, “On the feasibility of transfer-learning code smells using deep learning,” *arXiv preprint arXiv:1904.03031*, 2019.
- [62] T. Bryksin, E. Novozhilov, and A. Shpilman, “Automatic recommendation of move method refactorings using clustering ensembles,” in *Proceedings of the 2nd International Workshop on Refactoring*, 2018, pp. 42–45.
- [63] Z. Marian, G. Czibula, and I. G. Czibula, “Using software metrics for automatic software design improvement,” *Studies in Informatics and Control*, vol. 21, no. 3, p. 250, 2012.

- [64] Z. Marian, "A study on hierarchical clustering based software restructuring." *Studia Universitatis Babes-Bolyai, Informatica*, vol. 57, no. 2, 2012.
- [65] W. Pan, B. Jiang, and Y. Xu, "Refactoring packages of object-oriented software using genetic algorithm based community detection technique," *International journal of computer applications in technology*, vol. 48, no. 3, pp. 185–194, 2013.
- [66] A. Barbez, F. Khomh, and Y.-G. Guéhéneuc, "A machine-learning based ensemble method for anti-patterns detection," *Journal of Systems and Software*, vol. 161, p. 110486, 2020.