Automated Software Engineering (2025) 32:7
https://doi.org/10.1007/s10515-024-00476-3

Efficient feature envy detection and refactoring based on graph neural network

Dongjin Yu[1] · Yihang Xu[1] · Lehui Weng[1] · Jie Chen[1] · Xin Chen[1] · Quanxin Yang[1,2]

Received: 1 September 2023 / Accepted: 11 November 2024

Abstract
As one type of frequently occurring code smells, feature envy negatively affects class cohesion, increases coupling between classes, and thus hampers software maintainability. While progress has been made in feature envy detection, two challenges still persist. Firstly, existing approaches often underutilize method call relationships, resulting in suboptimal detection efficiency. Secondly, they lack the emphasis on feature envy refactoring, which is however the ultimate goal of feature envy detection. To address these challenges, we propose two approaches: **SCG** (**S**MOTE **C**all **G**raph) and **SFFL** (**S**ymmetric **F**eature **F**usion **L**earning). SCG transforms the feature envy detection problem into a binary classification task on a method call graph. It predicts the weights of edges, termed *calling strength*, to capture the strength of method invocations. Additionally, it converts the method-method call graph into a method-class call graph and recommends the smelly method to the external class with the highest *calling strength*. As a holistic approach focusing on refactoring feature envy directly, SFFL leverages four heterogeneous graphs to represent method-class relationships. Through Symmetric Feature Fusion Learning, it obtains representations for methods and classes. Link prediction is then employed to generate the refactored method-class ownership graph, which is regarded as the refactored results. Moreover, to address the limitations of existing metrics in accurately evaluating refactoring performance, we introduce three new metrics: $precision_2$, $recall_2$ and F_1-score$_2$. Extensive experiments on five open-source projects demonstrate the superiority of SCG and SFFL. The code and dataset used in our study are available at https://github.com/HduDBSI/SCG-SFFL.

Keywords Code smell · Feature envy · Refactoring · Graph neural network · Metrics

Extended author information available on the last page of the article

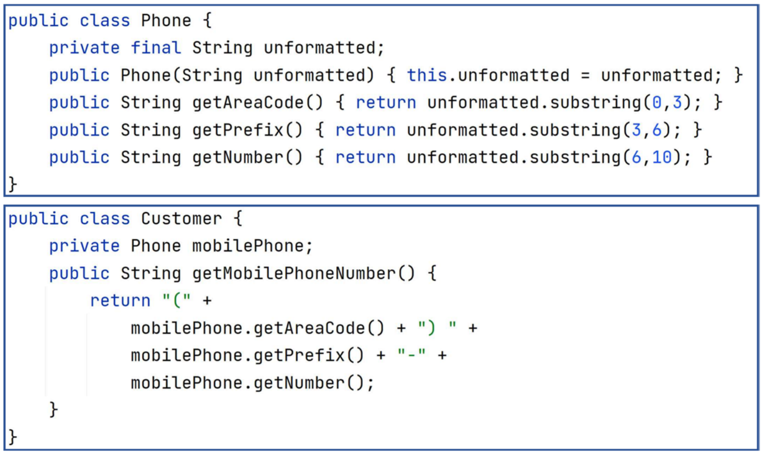
1 Introduction

In an ideal scenario, software developers strive to write efficient, readable, and maintainable code (Guo et al. 2021). However, in the real-world development process, developers often face the pressure of time and budgets, forcing them to prioritize short-term gains over long-term code quality (da S. Maldonado et al. 2017). Although these compromises may alleviate immediate pressures, they inevitably introduce Technical Debt (TD) (Cunningham 1993).

Code smells, one of the most serious forms of TD (Yu et al. 2022), refer to the confusing, complex, or even harmful design patterns in source code (Fowler 1999; Palomba et al. 2018). Two well-known code smells are *long method* and *god class*: the former refers to the method that is excessively long and performs multiple tasks, while the latter refers to the class that takes excessive responsibilities. These two kinds of code smells both violate the single responsibility principle (Shahidi et al. 2022), making the code harder to understand and maintain. Recent studies have highlighted the co-occurrence of code smells (Tsantalis et al. 2018) and found that *feature envy* is present in almost all *long method*s and *god class*es (Lozano et al. 2015).

As one type of the most common code smells, *feature envy* refers to a method that is more interested in an external class than its enclosing class (Fowler 1999). The interest of a method in a class is manifested in two aspects: 1) its invocation of methods in the class, and 2) its access to attributes of the class.

Figure 1 illustrates an instance of *feature envy*, where `get Mobile Phone Number` calls no methods within `Customer`, but three methods of `Phone`. It is evident that `getMobilePhoneNumber` exhibits *feature envy* towards the methods of `Phone` and would be better placed in `Phone`. Also, we can find that *feature envy* is a misplaced method, which reduces the cohesion of its own class and



```
public class Phone {
    private final String unformatted;
    public Phone(String unformatted) { this.unformatted = unformatted; }
    public String getAreaCode() { return unformatted.substring(0,3); }
    public String getPrefix() { return unformatted.substring(3,6); }
    public String getNumber() { return unformatted.substring(6,10); }
}
```

```
public class Customer {
    private Phone mobilePhone;
    public String getMobilePhoneNumber() {
        return "(" +
            mobilePhone.getAreaCode() + ") " +
            mobilePhone.getPrefix() + "-" +
            mobilePhone.getNumber();
    }
}
```

Fig. 1 An instance of *feature envy*. In this scenario, the class `Customer` has a method named `getMobilePhoneNumber`, which provides a North American-formatted mobile phone number by invoking excessive numbers of methods in another class `Phone`

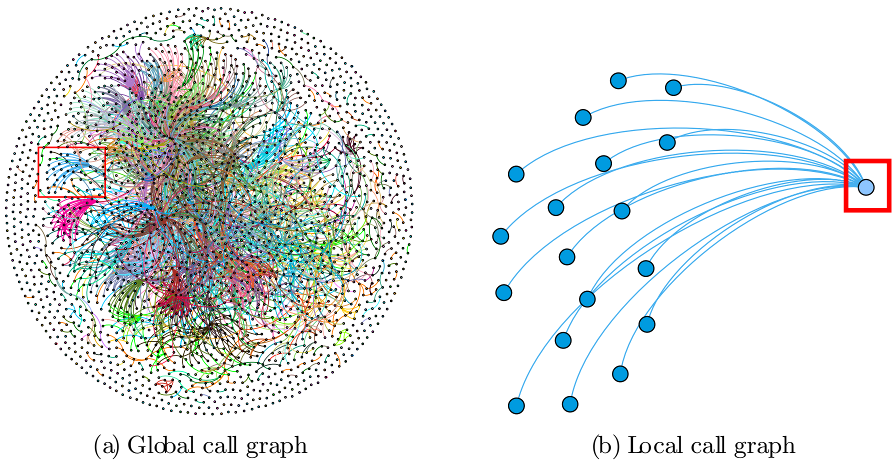


Fig. 2 Method call graph of Realm-java, where each node represents a method, each edge represents a method invocation, and methods belonging to the same class are depicted in the same color. In the local call graph, it is evident that the light-colored node is affected with *feature envy* as it exhibits calling relationships with 18 dark-colored nodes

increases the coupling between classes. One straightforward way to address this issue is to move the misplaced method to the class it is most interested in.

We argue that the method invocation relationship is the most significant manifestation of *feature envy*, which can be highly intricate within a project. To visualize these relationships, we utilize *Gephi*[1] to construct the method call graph of Realm-java as an example, which is shown in Fig. 2.

However, existing approaches (Guo et al. 2019; Wang et al. 2020; Yin et al. 2021; Zhao et al. 2022) fail to exploit the fundamental method call relationships. Specifically, they simply use manually designed expressions to convert method invocation information into distances, resulting in unsatisfactory detection performance. Additionally, the input format they adopt isolates potentially related samples, leading to poor refactoring capability .

In this paper, we present **SCG** (**S**MOTE **C**all **G**raph), a Graph Neural Network (GNN)-based approach, to address the problems of detecting and refactoring *feature envy*. SCG formulates the detection task as a node binary classification problem on the method call graph. It introduces the concept of *calling strength* to quantify the strength of method invocations in the call graph. Additionally, SCG converts the method call graph into a method-class call graph representation. Finally, it recommends moving the smelly method to the external class with the highest *calling strength*.

Furthermore, to integrate the processes of detecting and refactoring *feature envy*, we propose a holistic approach named **SFFL** (**S**ymmetric **F**eature **F**usion **L**earning) to address the *feature envy* refactoring problem. We collect

[1] https://gephi.org/

invocation, ownership, position, and semantic information of a project, and encode them into four directed heterogeneous graphs, where the nodes represent methods and classes, and the edges represent the invocation or ownership relationships between them. SFFL employs Symmetric Feature Fusion Learning to obtain representations of methods and classes. Finally, we introduce link prediction to generate the adjacency matrix representing the ownership relationships between methods and classes.

Our extensive experiments have shown that SCG and SFFL have their own advantages and could complement each other in different situations. In general, SFFL exhibits better detection and refactoring performance than SCG. However, when the training samples are extremely imbalanced, the detection performance of SCG is superior to SFFL. In cross-project scenarios, SCG achieves excellent detection and reconstruction performance with the less fine-tuning data (5%), while SFFL requires a moderate amount of fine-tuning data (10%) to achieve good detection and refactoring performance.

This work is the extension of our previous one (Yu et al. 2022) published at ISSRE 2022 with significant improvements. Our previous work proposes a novel approach named SCG based on Graph Neural Network (GNN), which effectively captures the pattern of *feature envy* and recommends its refactoring through *calling strength*. The main contributions of this work over our previous one are as follows:

- A holistic approach named SFFL is introduced based on the reconstruction of method-class ownership heterogeneous graph, which bypasses the detection step but reveals the essence of *feature envy* refactoring.
- Three new evaluation metrics are proposed, which have been proved to be effective for both *feature envy* detection and refactoring.
- Extensive experiments on five open-source projects demonstrate the superiority of SCG and SFFL over three competitors.

The rest of the paper is organized as follows. After Sect. 2 reviews the related work on *feature envy*, Sect. 3 describes our proposed approach SCG and Sect. 4 describes the holistic approach SFFL. We then discuss the traditional evaluation metrics and define new ones in Sect. 5. Next, we show the dataset and experimental settings in Sect. 6. Besides, we evaluate SCG and SFFL in Sect. 7 and Sect. 8, respectively. In Sect. 9, we discuss the threats to the validity of our approaches. Finally, we conclude our paper and explore the future work in Sect. 10.

2 Related work

In this section, we review the achievements made by researchers on *feature envy* detection and refactoring. We find that metrics-based approaches have shown effectiveness in both detecting and refactoring, while deep learning-based approaches have faced challenges in the refactoring phase.

2.1 Metrics-based approaches

Simon et al. (2001) introduced the first approach for *feature envy* detecting and refactoring. They formulated a distance measurement to characterize the cohesion between two entities (methods and attributes) e_i and e_j:

$$\text{distance}(e_i, e_j) = 1 - \frac{\left|p(e_i) \cap p(e_j)\right|}{\left|p(e_i) \cup p(e_j)\right|}. \tag{1}$$

Here, if e_i is a method, $p(e_i)$ denotes the method itself, as well as all methods and attributes accessed by it. If e_i is an attribute, $p(e_i)$ denotes the attribute itself and all methods that access it. Therefore, if a method has a closer distance to entities of an external class compared to its enclosing class, it indicates the presence of *feature envy* and suggests refactoring the method to the external class.

Tsantalis and Chatzigeorgiou (2009) built upon the work of Simon et al. (2001) and further proposed distance calculation formulas between methods and classes. If method m does not belong to class c, the distance between them is computed as follows:

$$\text{distance}(m, c) = 1 - \frac{|p(m) \cap p(c)|}{|p(m) \cup p(c)|}, p(c) = \left\{m_i \mid m_i \in c\right\}. \tag{2}$$

If m belongs to c, the distance between them is computed as

$$\text{distance}(m, c) = 1 - \frac{|p(m) \cap p'(c)|}{|p(m) \cup p'(c)|}, p'(c) = p(c) \setminus m. \tag{3}$$

By calculating the distances between a method and all classes, it is possible to determine whether the method exhibits a smell and to which class it should be moved. The approach is implemented in a well-known code smell detection tool called *JDeodorant*, which is one of the competitors in this paper.

Sales et al. (2013) proposed the concept of method dependency, which has a broader range than the entity used by (Simon et al. 2001; Tsantalis and Chatzigeorgiou 2009). It not only includes method calls and field accesses, but also considers local declarations, return types and so on. The similarity between two methods, m_1 and m_2, is defined as

$$\begin{aligned} \text{similarity}(m_1, m_2) &= \frac{a}{a + 2(b + c)}, \\ \text{where } a &= \left|d(m_1) \cap d(m_2)\right|, \\ b &= \left|d(m_1) \setminus d(m_2)\right|, \\ c &= \left|d(m_2) \setminus d(m_1)\right|. \end{aligned} \tag{4}$$

Here, $d(m_i)$ denotes the set of the dependencies of m_i. Furthermore, the similarity between method m and class c can be calculated as:

$$\text{similarity}(m, c) = \frac{1}{|c|} \sum_{m_i \in c} \text{similarity}(m, m_i), \tag{5}$$

where $|c|$ represents the number of methods in class c. This approach is implemented by *JMove* (Terra et al. 2018), which is also one of the competitors in this paper.

2.2 Deep learning-based approaches

Metrics-based approaches essentially calculate the similarity between methods and classes by counting the number of shared and distinct code features they possess, which are easy to implement but often do not yield satisfactory results. Consequently, researchers have turned to deep learning models to improve the detection and refactoring of *feature envy*.

Liu et al. (2018) incorporated the distance proposed by Tsantalis and Chatzigeorgiou (2009) and considered the textual similarity of method names and class names. They employed a Convolutional Neural Network (CNN) (LeCun et al. 1998) and defined the input as follows:

$$\text{input} =< \text{name}_m, \text{name}_{ec}, \text{name}_{tc}, \text{dist}(m, ec), \text{dist}(m, tc) > \tag{6}$$

where m is a method, ec is its enclosing class, tc is a potential target class, dist is defined in Eqs. (2, 3). If the model output likelihood is higher than 0.5, they suggest m has *feature envy*, and should be moved to tc with the highest likelihood. Liu's work is also one of the competitors in this paper.

Other researchers have also employed deep learning models for *feature envy* detection and refactoring. Guo et al. (2019) employ Attention Mechanism (Bahdanau et al. 2015) and LSTM (Hochreiter and Schmidhuber 1997) to construct the semantic representations for methods; Wang et al. (2020) utilize BiLSTM (Schuster and Paliwal 1997) to obtain semantic and distance representations, and use Self-Attention Mechanism (Vaswani et al. 2017) to focus on more important features; Yin et al. (2021) further divide semantic information and metrics information into global and local features, using LSTM and CNN (LeCun et al. 1998) for learning; Zhao et al. (2022) use feature mining model and dual attention to obtain comprehensive representations of entities.

While these deep learning-based approaches for *feature envy* detection have become increasingly complex, researchers (Guo et al. 2019; Wang et al. 2020; Yin et al. 2021; Zhao et al. 2022) still adopt the input format defined by Liu et al. (2018).

This is primarily because Liu et al. provide a dataset for *feature envy*, which includes both textual and pre-computed distance information. However, we argue that this dataset has three main issues, as discussed in Sect. 9.1.

Furthermore, it is worth noting that a certain number of studies (Sharma et al. 2021; Guo et al. 2019; Zhang and Jia 2022; Yin et al. 2021; Zhao et al. 2022) do not consider *feature envy* refactoring. This can be attributed to the limitations of the existing input format, which restricts the ability to capture and incorporate the necessary information required for the refactoring process. Specifically, the existing input format is unable to simultaneously consider all the classes accessed by the smelly method during refactoring, resulting in a low accuracy in refactoring recommendations.

As the extension of our previous paper published at ISSRE (Yu et al. 2022), this paper presents the approaches to detecting and refactoring *feature envy* based on Graph Neural Network (GNN). In particular, it focuses on the essence of *feature envy* and considers *feature envy* refactoring as a fundamental problem of edge reconstruction in a heterogeneous graph. In such a graph, the nodes represent methods and classes, while the edges represent the ownership relationships between methods and classes.

3 The approach of SCG

In this section, we introduce a novel approach called **SCG** (**S**MOTE **C**all **G**raph) for the detection and refactoring of *feature envy*.

3.1 Overview

SCG is proposed based on the following three considerations.

Firstly, the existing deep learning-based approaches (Liu et al. 2018; Wang et al. 2020; Zhao et al. 2022; Guo et al. 2019; Yin et al. 2021; Zhang and Jia 2022) fail to fully utilize the method invocation information. Instead, they simply use manually designed expressions to convert method invocation information into distances, which are then used as inputs to the model. However, we believe that method invocation information, as one of the primary manifestations of *feature envy*, should not be processed or transformed in such a way because it would lead to the loss of other important information.

Secondly, these approaches do not view the problem from a graph perspective. They treat samples independently without considering their interconnections. For example, assuming there are three potential target classes, tc_1, tc_2 and tc_3 for a method m, and m is located in class ec. Thus, they make three samples defined as:

$$\text{sample}_1 = < \text{name}_m, \text{name}_{ec}, \text{name}_{tc_1}, \text{dist}(m, ec), \text{dist}(m, tc_1) >, \tag{7}$$

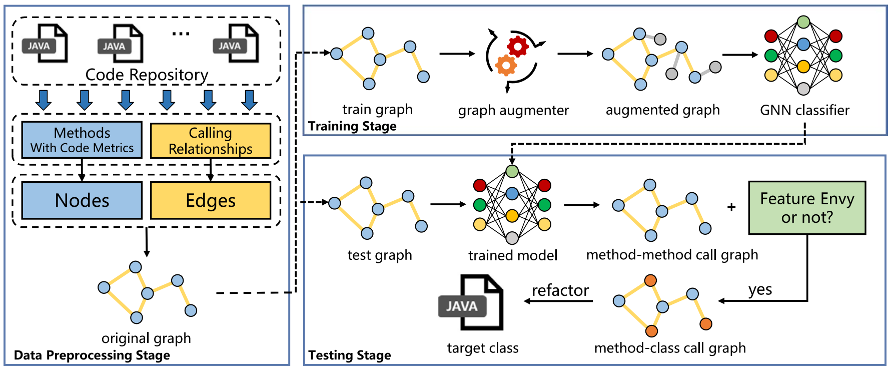


Fig. 3 Overview of SCG

$$\text{sample}_2 =< \text{name}_m, \text{name}_{ec}, \text{name}_{tc_2}, \text{dist}(m, ec), \text{dist}(m, tc_2) >, \tag{8}$$

$$\text{sample}_3 =< \text{name}_m, \text{name}_{ec}, \text{name}_{tc_3}, \text{dist}(m, ec), \text{dist}(m, tc_3) > . \tag{9}$$

In other words, these samples are treated independently in the model, and consequently do not share any semantic or distance information. This limitation leads to the model's inability to determine how many external classes a method calls.

Lastly, the current approaches lack a holistic view to refactoring. Indeed, when the input samples are treated independently, the output probabilities are also independent. This creates a challenge when a smelly method needs to be refactored into multiple external classes. The decision of which external class to move a method to, based on independent probabilities, is obviously controversial and limited.

Therefore, we propose SCG, which considers *feature envy* detection from the perspective of the method-method call graph. Figure 3 presents an overview of SCG, which can be divided into three stages. In the **Data Preprocessing Stage**, we collect code metrics and calling relationships for each method in a Java project. We then convert them into nodes and edges of a graph, where each node represents a method with its code metrics and each edge represents the calling relationship between two methods. In the **Training Stage**, to avoid the unbalance of positive and negative samples, a graph augmenter is introduced to obtain the augmented graph, which is then fed into a GNN classifier for training. In the **Test Stage**, the well-trained model tells whether a method contains *feature envy* or not. The model also outputs a weighted method-method call graph, which is then transformed into a method-class call graph. Based on the edge weight (*calling strength*) of the method-class call graph, we recommend moving a method, which is predicted smelly, to the class with the highest *calling strength* to it.

Table 1 Code metrics description

Type	Name	Description
Structural	LOC	Lines of code
	CC	Cyclomatic complexity
	PC	Parameters count
Invocation	NCMEC	Number of calls to methods of external classes
	NCMIC	Number of calls to methods of its class
	NECA	Number of external classes accessed
	NAMFAEC	Number of accesses to most frequently accessed external class

3.2 Feature collection

3.2.1 Calling relationships

We believe that the calling relationship is very important for *feature envy* detection. Therefore, we leverage *SciTools Understand*[2] to obtain the set of calling relationships between methods in a software project. The calling relationships provided by this tool also include the classes to which both the caller and the callee belong.

It is important to acknowledge that some methods are dynamically invoked, and the specific method being called can only be determined at runtime. In these cases, *SciTools Understand* will capture all potential calling relationships. While the analysis results from *SciTools Understand* may not be highly precise, they are reasonably comprehensive in capturing all potential calling relationships. Therefore, we believe that this moderate analysis of calls has minimal impact on the performance of our proposed approaches.

It should also be noted that if the called method does not belong to the project, we will simply remove it from the set of calling relationships. The main reason is that almost all methods may establish calling relationships with the methods of the primary package (such as `java.lang` and `java.util`), which however have no actual value for detection and refactoring.

3.2.2 Code metrics

For each method in a software project, we calculate its structural and invocation information. The definitions of these metrics are given in Table 1.

We select three common structural metrics, i.e., Lines Of Code (LOC), Cyclomatic Complexity (CC) and Parameters Count (PC), to describe the structural information of the method.

Since the main manifestation of *feature envy* is that a method calls the methods of an external class too often, we should also consider calling information of the method. Based on the calling relationships provided by *SciTools Understand*, we calculate four metrics for each method: the number of times it calls the methods of external classes, the number of times it calls the methods of its enclosing class, the number of external

[2] https://emenda.com/scitools-understand/

classes it accesses, and the number of times it accesses the external class being accessed most frequently.

3.3 Graph representation

The aim of graph representation is to model the methods with code metrics and the calling relationships between methods as nodes and edges in the graph. In a normal GNN model, the graph can be represented as

$$\mathbf{G} = (\mathbf{V}, \mathbf{E}, \mathbf{F}), \tag{10}$$

where **V** represents all nodes in the graph, **E** represents a collection of edges between nodes, and **F** presents the feature matrix of nodes. In this paper, the actual contents of **V**, **E** and **F** are as follows:

- $\mathbf{V} = \{v_1, v_2, ..., v_n\}$ is the set of n nodes, where each node represents a method with a unique identifier v_i.
- **E** is a collection of edges, where each edge represents a calling relationship between two methods. Therefore, each edge is denoted as $e =< v_i, v_j >$.
- **F** represents the feature matrix composed of feature vectors of each method. Here, the feature vector is denoted as

$$\vec{f} =< \text{LOC}, \text{CC}, \text{PC}, \text{NCMEC}, \text{NCMIC}, \text{NECA}, \text{NAMFAEC} > . \tag{11}$$

Also, we can obtain the adjacency matrix of the graph, which is defined as:

$$\mathbf{A}[i,j] = \begin{cases} 1 \text{ if } < v_i, v_j > \in \mathbf{E}, \\ 0 \text{ if } < v_i, v_j > \notin \mathbf{E}. \end{cases} \tag{12}$$

3.4 Graph augmentation

In our specific scenario, the number of smelly methods is significantly smaller than that of normal methods. Under such a circumstance, training a GNN classifier with the original graph will underestimate the methods with smells, leading to suboptimal performance inevitably. Therefore, it is necessary to apply a data balancing technique before training.

Though the traditional oversampling techniques are easy to implement, their effects are not ideal for a GNN classifier (Zhao et al. 2021). In our approach, we exploit GraphSMOTE (Zhao et al. 2021) as the graph augmenter to generate the smelly nodes (i.e., methods with feature envy) and edges (i.e., calling relationship associated with smelly methods) connected to these nodes. Figure 4 illustrates the framework of the graph augmenter used in our approach.

The graph augmenter consists of the following three components:

- *Feature extractor*, which transforms node attributes and local graph topology into a high-dimensional embedding space.

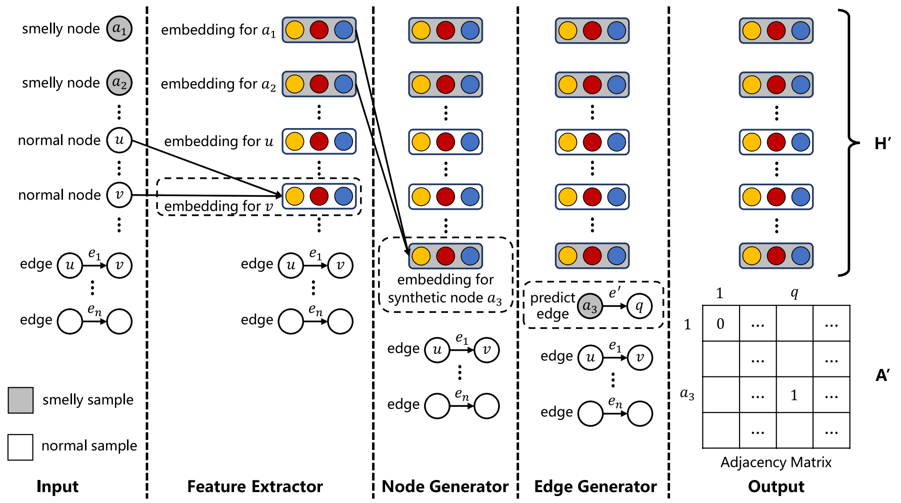


Fig. 4 Framework of graph augmenter

- *Node generator*, which employs an oversampling technique to generate synthetic nodes with smells based on the node embedding representations.
- *Edge generator*, which is trained on the original graph to predict the associations between the synthetic smelly nodes and the original graph.

Finally, the graph augmenter outputs the node embedding representations and the adjacency matrix representing the connection relationship between nodes. The implementation details of the three components are described below.

3.4.1 Extracting feature

Most of the previous oversampling techniques synthesize minority samples directly in the original feature space, and generate minority samples by interpolating between two similar samples of the same class (Chawla et al. 2002). In our approach, however, nodes are interconnected in the graph, which indicates that directly exploiting the traditional oversampling techniques may lose the relationship information between nodes. Therefore, before generating synthetic nodes, we need to learn the representations of nodes in the graph, such as node attributes, labels and local graph topology, so as to better analyze the similarity between nodes.

We employ GraphSAGE (Hamilton et al. 2017) as the feature extractor to learn the representations of nodes. Different from the direct push framework that can only learn embedding representations of a fixed graph, GraphSAGE is an inductive framework, which can effectively learn the graph topology information and node embedding representations according to the change in the neighbor relationships of the nodes.

In order to get the embedding of node v, we first obtain the embedding representation of its neighbors, which is defined as follows:

$$\vec{h}^{1}_{\mathcal{N}(v)} = \mathrm{MEAN}\Big(\Big\{\vec{f}_u, \forall u \in \mathcal{N}(v)\Big\}\Big), \tag{13}$$

where $\vec{f}_u$ represents the feature vector of node u, $\mathcal{N}(v)$ represents the set of neighbor nodes of node v. Further, we can obtain the embedding of the node v:

$$\vec{h}^{1}_{v} = \mathrm{ReLU}\Big(\mathbf{W}^1 \cdot \mathrm{CONCAT}(\vec{f}_v, \vec{h}^{1}_{\mathcal{N}(v)})\Big), \tag{14}$$

where $\vec{h}^1_v$ is the embedding representation of node v, $\mathbf{W}^1$ represents a trainable weight matrix, CONCAT represents the vector concatenating operation, and ReLU represents an activation function.

For each node in the graph, the feature extractor concatenates the features of the node with the average features of its neighboring nodes, and converts them into the representation in the high-dimensional embedding space through nonlinear transformation. For simple representation, we denote this high-dimensional embedding space as

$$\mathbf{H} = \Big\{\vec{h}^1_1, \vec{h}^1_2, ..., \vec{h}^1_n\Big\}. \tag{15}$$

Specifically, as shown in Fig. 4, assuming that node v only has one neighbor node u in the original graph, the feature extractor generates the representation of node v based on the features of node u and node v.

3.4.2 Generating smelly nodes

After the feature extractor learns the information of neighboring nodes, we can obtain the embedding of each node from $\mathbf{H}$ and implement oversampling on this basis. Referring to the implementation of SMOTE (Chawla et al. 2002), the process can be divided into three steps:

- An oversampling ratio N is set according to the sample imbalance ratio.
- For each smelly node embedding $\vec{h}^1_v$, N nearest smelly node embeddings are selected based on Euclidean distance.
- For each selected node embedding $\vec{h}^1_u$, a synthetic smelly node embedding is generated:

$$\vec{h}^1_{v,u} = (1-\delta)\cdot\vec{h}^1_v + \delta\cdot\vec{h}^1_u, \tag{16}$$

  where $h^1_{u,v}$ is the embedding of the synthetic node, and δ is a random number following uniform distribution in the range [0, 1].

As $\vec{h}^1_v$ and $\vec{h}^1_u$ are both the embeddings of smelly nodes, their representations are very close to each other, and the synthetic node should also have smell. Specifically, as shown in Fig. 4, the node generator generates the embedding of smelly node a_3 based on the embeddings of smelly nodes a_1 and a_2.

3.4.3 Generating edges

Since the synthetic smelly nodes are not related to the original graph, if these synthetic nodes are directly put into training, it will reduce the performance of the classifier. Therefore, we train an edge generator on the original graph, and exploit it to predict the relationships between the synthetic nodes and the original graph.

Firstly, the edge generator obtains the adjacency matrix $\mathbf{A}$ representing the connection relationships between nodes, and then models the existence of edges between nodes. It trains on real nodes and edges to learn the relationships between nodes, and predicts the neighbors' information for the generated synthetic nodes. The predicted edge is defined as follows:

$$\hat{\mathbf{A}}[v,u] = \text{sigmoid}\left(\mathbf{W}\vec{h}_v^1 \cdot (\mathbf{W}\vec{h}_u^1)^T\right), \tag{17}$$

where $\hat{\mathbf{A}}[v,u]$ is the predicted edge weight between nodes v and u, $\vec{h}_v^1$ is the embedding of node v, $\vec{h}_u^1$ is the embedding of node u, $\mathbf{W}$ is a trainable parameter matrix to capture the interaction between these two nodes, and sigmoid is used to transform the input into probability value ranging from 0 to 1. The loss function of the edge generator is defined as:

$$\mathcal{L}_{edge} = \|\mathbf{A} - \hat{\mathbf{A}}\|^2, \tag{18}$$

where $\hat{\mathbf{A}}$ represents the predicted adjacency matrix of the original graph, and $\mathbf{A}$ represents the actual adjacency matrix of the original graph. With the edge generator, we obtain the augmented adjacency matrix:

$$\mathbf{A}'[v',u] = \begin{cases} 1 \text{ if } \hat{\mathbf{A}}[v',u] > 0.5, \\ 0 \text{ otherwise,} \end{cases} \tag{19}$$

where v' represents the synthetic smelly node, u represents the normal node, $\hat{\mathbf{A}}[v',u]$ denotes the predicted relationship between nodes v' and u. More specifically, as shown in Fig. 4, the edge generator predicts that there is a calling relationship between the original node q and the synthetic node a_3. Consequently, the two nodes are connected through generated edge e'.

Indeed, the predicted adjacency matrix $\hat{\mathbf{A}}$ can provide a more granular characterization of the calling relationships between nodes. It can be instrumental in assisting the refactoring of smelly nodes, which will be discussed in Sect. 3.6.

Finally, these generated nodes and edges will be added to the original graph, and the updated node embedding space $\mathbf{H}'$ and adjacency matrix $\mathbf{A}'$ will serve as the input of the GNN-based classifier.

3.5 GNN classifier

This section introduces how to construct a GNN-based classifier for *feature envy* detection. As Fig. 5 indicates, the classifier is divided into five layers defined as follows:

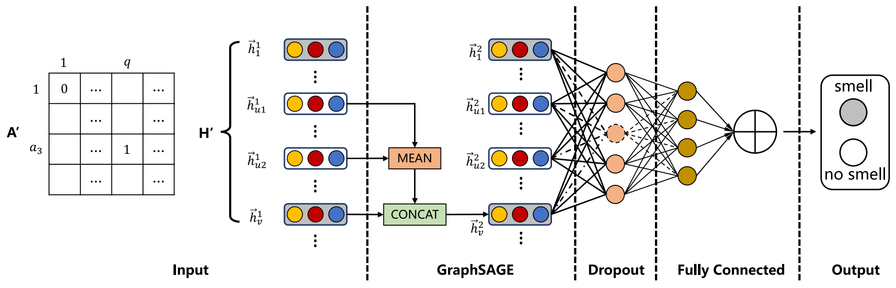


Fig. 5 Structure of the GNN classifier

- *Input layer* We use the outputs of the graph augmenter as inputs to the GNN (Graph Neural Network) classifier. Here, $\mathbf{H}'$ is the embeddings of all nodes (including real nodes and synthetic nodes), and $\mathbf{A}'$ is the adjacency matrix representing the connection relationship between nodes in the augmented graph.
- *GraphSAGE layer* Similar to the GraphSAGE (Hamilton et al. 2017) mentioned in Sect. 3.4.1, this layer further updates the node embeddings. For each node v, it first obtains the set of its neighboring nodes, and then calculates the mean embedding of the neighboring nodes defined as:

$$\vec{h}^2_{\mathcal{N}(v)} = \mathrm{MEAN}(\left\{\vec{h}^1_u, \forall u \in \mathcal{N}(v)\right\}). \tag{20}$$

 Next, the mean embedding is concatenated with the embedding of node v and the nonlinear transformation is carried out to update the embedding of node v. The process is shown in Eq. (21):

$$\vec{h}^2_v = \mathrm{ReLU}\left(\mathbf{W}^2 \cdot \mathrm{CONCAT}(\vec{h}^1_v, \vec{h}^2_{\mathcal{N}(v)})\right), \tag{21}$$

 where $\vec{h}^2_v$ represents the embedding of node v after updating, $\vec{h}^1_v$ represents the embedding of node v before updating, $\mathbf{W}^2$ is a trainable weight matrix, and CONCAT represents the vector concatenating operation. Specifically, as shown in Fig. 5, it is assumed that in the augmented graph, both nodes u_1 and u_2 have edges pointing to node v. Therefore, when updating the embedding of node v, it first calculates the mean embedding of u_1 and u_2 in the embedding space $\mathbf{H}'$, and then concatenates the mean embedding with the embedding of node v to update the embedding of node v.
- *Dropout layer* The function of this layer is to prevent the classifier from falling into overfitting during node classification training. In the process of model training, the layer temporarily discards some hidden neurons from the network with a probability of 10%. The purpose of this is to reduce the synergy between hidden layer nodes, so that the model will not rely too much on local features. In this way, the generalization ability of the model can be improved.
- *Fully connected layer and output layer* The fully connected layer takes the output of the dropout layer as the input and converts it into a two-dimensional vector for final classification. The output layer receives the vector and outputs

the prediction result of the classifier through the activation function softmax. The predicted result can be expressed as:

$$[1 - p_v, p_v] = \text{softmax}\left(\mathbf{W}^3 \vec{h}_v^2\right), \tag{22}$$

where $\mathbf{W}^3$ is a trainable weight matrix, and p_v is the probability that the node v is predicted smelly. We use the cross entropy-loss function during model training, which can be defined as:

$$\mathcal{L}_{node} = -\frac{1}{N} \sum_{v}^{N} \left(y_v \cdot \log(p_v) + (1 - y_v) \cdot \log(1 - p_v)\right), \tag{23}$$

where y_v represents the real label of node v (e.g., 1 means the node has a smell). In addition, since the performance of the model depends on the quality of the edges generated by the graph augmenter, inspired by literature (Zhao et al. 2021), we combine the loss functions of the graph augmenter and GNN classifier. The final loss of the GNN model is computed as:

$$\mathcal{L} = \mathcal{L}_{node} + \lambda \mathcal{L}_{edge}, \tag{24}$$

where λ is a parameter that adjusts the contribution of the graph augmenter and the GNN classifier to the final result.

3.6 Refactoring recommendation

For methods that are predicted to have smells, we should provide refactoring suggestions to move them to the classes that best fit their functional implementation.

Since the final loss function in Eq. (24) considers both node classification's loss and edge prediction's loss, the well-trained model has a very deep understanding of the calling relationships of all methods. In details, the calling relationship between two methods should not only describe who calls who, but also describe the *calling strength* of one method to another. As a result, we exploit the trained edge generator to predict the *calling strength* between methods. Next, we convert the *calling strength* between methods to the *calling strength* between methods and classes:

$$\mathbf{R}[m, c] = \sum_{m_i \in c} \hat{\mathbf{A}}[m, m_i], \tag{25}$$

where $\mathbf{R}[m, c]$ represents the *calling strength* between method m and class c, m_i is the method located in c, and $\hat{\mathbf{A}}[m, m_i]$ is the predicted *calling strength* between m_i and m.

For each method predicted to have *feature envy*, we calculate its *calling strength* with other external classes and recommend moving it to the external class with the highest *calling strength*.

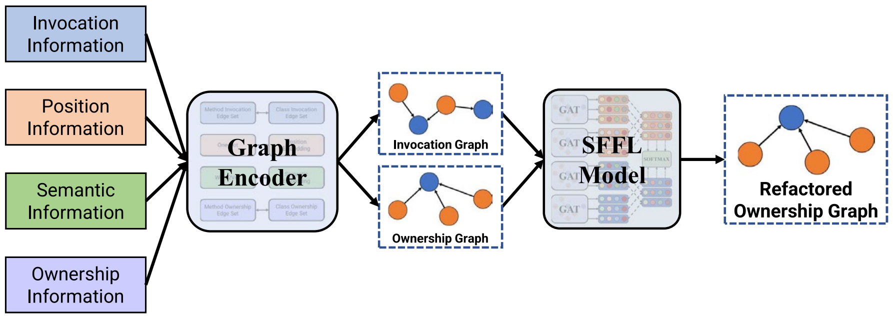


Fig. 6 Overview of SFFL

4 The holistic approach: SFFL

In this section, considering that SCG may not fully utilize heterogeneous information, we propose a holistic approach named **SFFL** (**S**ymmetric **F**eature **F**usion **L**earning). Different from SCG, SFFL treats the processes of detecting and refactoring feature envy in an integrated way.

4.1 Overview

We propose SFFL based on the following three considerations.

Firstly, the previous deep learning-based approaches for *feature envy* primarily focus on detection while ignoring refactoring (Sharma et al. 2021; Zhao et al. 2022; Guo et al. 2019; Yin et al. 2021; Zhang and Jia 2022). Some of these approaches (Liu et al. 2018; Wang et al. 2020) consider both detection and refactoring tasks, but prioritize detection performance during the modeling process (i.e., the model only fits binary classification labels). They adopt a refactoring strategy after training, converting binary results into multi-class results. However, we argue that *feature envy* refactoring is actually the goal of its detection, and should be paid more attention.

Secondly, it is challenging to introduce the refactoring loss in SCG. The loss function in SCG consists of node loss and edge loss, adjusted by a weight parameter. If the refactoring loss is further introduced, the weight relationships among the three components need to be considered together, which leads to a complex problem requiring numerous experiments to search for optimal weights.

Lastly, using a heterogeneous graph for *feature envy* refactoring is feasible. In the process of refactoring in SCG, we transform the *calling strength* between methods into the *calling strength* between methods and classes, achieving the conversion from a method-method call graph to a method-class call graph. This is actually the inspiration for SFFL. Since SCG inevitably needs to consider the method-class heterogeneous graph during refactoring, it is straightforward to use a heterogeneous graph directly to handle *feature envy* detection and refactoring.

Therefore, we propose SFFL, which simultaneously considers detection and refactoring during model training (i.e., the model directly fits multi-class labels), making the model more comprehensive and holistic. Figure 6 illustrates an overview of SFFL. Initially, for a Java project, we collect information about methods and classes on invocation, position, semantics and ownership. Subsequently, we input these four categories of information into a graph encoder to obtain call graphs and ownership graphs for methods and classes. These graphs are then fed into a GNN-based Symmetric Feature Fusion Learning (SFFL) model to obtain the refactored ownership graph for methods and classes.

4.2 Data collection

4.2.1 Invocation information

We collect the invocation information between methods, and the detailed process can be found in Sect. 3.2.1.

4.2.2 Ownership information

In addition to knowing the method invocation relationships, refactoring *feature envy* also requires knowing which class a method belongs to. If the caller and the callee belong to the same class, it is obvious that no refactoring is needed. Conversely, if the caller and the callee do not belong to the same class, the caller may be a potentially smelly method. Another purpose of ownership information is to indicate the direction for refactoring (i.e., the smelly caller should be moved to the class to which the callee belongs).

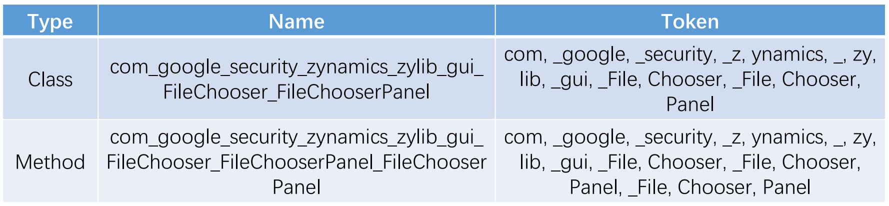
4.2.3 Semantic information

The similarity in semantics between method names and class names to some extent indicates *feature envy* (Liu et al. 2018). We collect method names and class names, including their paths. We believe that the package's path name, which includes deeper levels of semantic information, can provide insights into the functional modules to which a class or a method belongs. The reason lies in that the package's path name in software development contains valuable developers' insights into the functional modules of classes and methods.

We tokenize these names using *tiktoken*,[3] which is currently one of the most popular tokenizer tools with a segmentation speed that is 3–6 times faster than that of Hugging Face.[4] Figure 7 shows the tokenization results of class `FileChooserPanel` and its constructor. We discuss the adoption of *tiktoken* and its effectiveness in Sect. 9.2.

[3] https://github.com/openai/tiktoken

[4] https://github.com/huggingface/tokenizers



Type	Name	Token
Class	com_google_security_zynamics_zylib_gui_ FileChooser_FileChooserPanel	com, _google, _security, _z, ynamics, _, zy, lib, _gui, _File, Chooser, _File, Chooser, Panel
Method	com_google_security_zynamics_zylib_gui_ FileChooser_FileChooserPanel_FileChooser Panel	com, _google, _security, _z, ynamics, _, zy, lib, _gui, _File, Chooser, _File, Chooser, Panel, _File, Chooser, Panel

Fig. 7 Examples of tokenization results

4.2.4 Position information

To transform the aforementioned information into a graph structure, we need to assign appropriate identifiers to methods and classes. We start from 0 and assign unique identifiers to methods and classes, respectively. To convert the invocation and ownership information into corresponding identifiers, we use the format $<\text{caller\_id}, \text{callee\_id}>$ for method invocations and $<\text{method\_id}, \text{class\_id}>$ for ownership relationships.

4.3 Graph encoder

As shown in Fig. 8, we use a graph encoder to transform the aforementioned information into heterogeneous graphs, denoted as

$$\mathbf{G} = (\mathbf{V}_\mathrm{m}, \mathbf{V}_\mathrm{c}, \mathbf{E}, \mathbf{F}_\mathrm{m}, \mathbf{F}_\mathrm{c}), \tag{26}$$

where $\mathbf{V}_\mathrm{m} = \{v_1, v_2, ..., v_m\}$ is the set of m method nodes, $\mathbf{V}_\mathrm{c} = \{v_1, v_2, ..., v_c\}$ is the set of c class nodes, $\mathbf{E} = \{e_1, e_2, ..., e_k\}$ is the set of edges, $\mathbf{F}_\mathrm{m} \in \mathbb{R}^{m\times d}$ is the feature matrix of m method nodes with d features, and $\mathbf{F}_\mathrm{c} \in \mathbb{R}^{m\times d}$ is the feature matrix of c class nodes with d features.

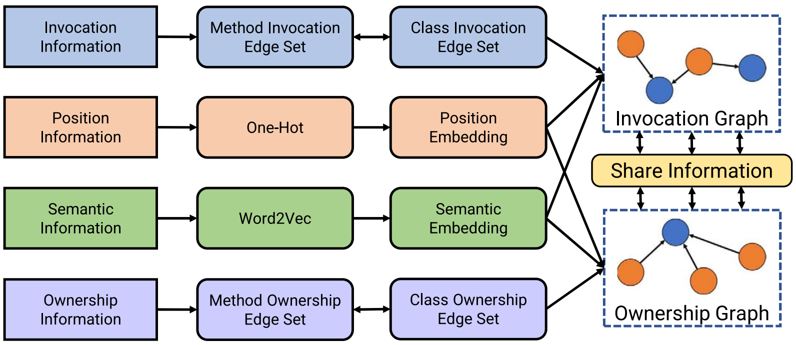


Fig. 8 Graph encoder

4.3.1 Ownership edge set

Based on the position information, we can represent the ownership relationship between a method and a class as a directed edge:

$$e_{\mathrm{mc\_own}} =< \mathrm{method\_id}, \mathrm{class\_id} > . \tag{27}$$

Further, we can obtain the edge set $\mathbf{E}_{\mathrm{mc\_own}}$, where each edge represents which class a method belongs to. By performing the reverse operation on each edge in $\mathbf{E}_{\mathrm{mc\_own}}$, that is

$$e_{\mathrm{cm\_own}} =< \mathrm{class\_id}, \mathrm{method\_id} >, \tag{28}$$

we can obtain the edge set $\mathbf{E}_{\mathrm{cm\_own}}$, where each edge represents which method a class owns.

4.3.2 Invocation edge set

Based on the position information, we can transform each method invocation into a directed edge:

$$e_{\mathrm{mm\_call}} =< \mathrm{caller\_id}, \mathrm{callee\_id} > . \tag{29}$$

Further, we can obtain the edge set $\mathbf{E}_{\mathrm{mm}}$, where each edge represents the invocation relationship between two methods. By performing the transitive operation (i.e., ensuring callee_id is the same as method_id), we obtain

$$e_{\mathrm{mc\_call}} = e_{\mathrm{mm\_call}} \circ e_{\mathrm{mc\_own}} =< \mathrm{caller\_id}, \mathrm{class\_id} >, \tag{30}$$

which represents the relationship between a method and the class it calls. Furthermore, we can obtain the edge set $\mathbf{E}_{\mathrm{mc\_call}}$, where each edge represents which method accesses a class. By applying the same reverse operation on $\mathbf{E}_{\mathrm{mc\_call}}$, we obtain $\mathbf{E}_{\mathrm{cm\_call}}$, where each edge represents which class is accessed by a method.

4.3.3 Semantic embedding

We create a corpus by collecting the tokenized results of method names and class names. Using Word2Vec (Mikolov et al. 2013), a widely-used technique in natural language processing, we train word embeddings on the corpus. Word2Vec enables us to obtain a vector representation for each token of a method or class name .

To obtain the semantic embedding for each method or class name, we perform average pooling on the word vectors corresponding to its tokens. This process allows us to capture the overall semantic meaning of the name. As a result, we can obtain the method semantic embedding matrix $\mathbf{S}_{\mathrm{m}} \in \mathbb{R}^{m \times d}$ and the class semantic embedding matrix $\mathbf{S}_{\mathrm{c}} \in \mathbb{R}^{c \times d}$, where m represents the number of methods, c represents the number of classes, and d represents the vector size of the semantic embeddings.

4.3.4 Position embedding

We employ one-hot encoding to encode the position information of methods and classes, resulting in two identity matrices $\mathbf{I}_{\mathrm{m}} \in \mathbb{R}^{m \times m}$ and $\mathbf{I}_{\mathrm{c}} \in \mathbb{R}^{c \times c}$. To ensure that the dimension of the position embedding matches the dimension of the semantic embedding, we perform a linear transformation using two matrices that follow a uniform distribution, $\mathbf{W}_{\mathrm{m}} \in \mathbb{R}^{m \times d}$ and $\mathbf{W}_{\mathrm{c}} \in \mathbb{R}^{c \times d}$, respectively. Thus, the method position embedding matrix is computed as

$$\mathbf{P}_{\mathrm{m}} = \mathbf{I}_{\mathrm{m}} \times \mathbf{W}_{\mathrm{m}} \in \mathbb{R}^{m \times d}, \tag{31}$$

and the class position embedding matrix is computed as

$$\mathbf{P}_{\mathrm{c}} = \mathbf{I}_{\mathrm{c}} \times \mathbf{W}_{\mathrm{c}} \in \mathbb{R}^{c \times d}. \tag{32}$$

4.3.5 Graph representation

We merge method semantic embeddings and method position embeddings to obtain the method node feature matrix, which is computed as

$$\mathbf{F}_{\mathrm{m}} = \mathbf{S}_{\mathrm{m}} + \mathbf{P}_{\mathrm{m}} \in \mathbb{R}^{m \times d}. \tag{33}$$

Similarly, the class node feature matrix is computed as

$$\mathbf{F}_{\mathrm{c}} = \mathbf{S}_{\mathrm{c}} + \mathbf{P}_{\mathrm{c}} \in \mathbb{R}^{c \times d}. \tag{34}$$

Once node feature matrices and edge sets are in place, it is easy to obtain the four heterogeneous graphs:

$$\mathbf{G}_{\mathrm{mc\_call}} = \left\{ \mathbf{V}_{\mathrm{m}}, \mathbf{V}_{\mathrm{c}}, \mathbf{E}_{\mathrm{mc\_call}}, \mathbf{F}_{\mathrm{m}}, \mathbf{F}_{\mathrm{c}} \right\}, \tag{35}$$

$$\mathbf{G}_{\mathrm{cm\_call}} = \left\{ \mathbf{V}_{\mathrm{m}}, \mathbf{V}_{\mathrm{c}}, \mathbf{E}_{\mathrm{cm\_call}}, \mathbf{F}_{\mathrm{m}}, \mathbf{F}_{\mathrm{c}} \right\}, \tag{36}$$

$$\mathbf{G}_{\mathrm{mc\_own}} = \left\{ \mathbf{V}_{\mathrm{m}}, \mathbf{V}_{\mathrm{c}}, \mathbf{E}_{\mathrm{mc\_own}}, \mathbf{F}_{\mathrm{m}}, \mathbf{F}_{\mathrm{c}} \right\}, \tag{37}$$

$$\mathbf{G}_{\mathrm{cm\_own}} = \left\{ \mathbf{V}_{\mathrm{m}}, \mathbf{V}_{\mathrm{c}}, \mathbf{E}_{\mathrm{cm\_own}}, \mathbf{F}_{\mathrm{m}}, \mathbf{F}_{\mathrm{c}} \right\}. \tag{38}$$

Since $\mathbf{G}_{\mathrm{mc\_call}}$ and $\mathbf{G}_{\mathrm{cm\_call}}$ can be converted to each other, $\mathbf{G}_{\mathrm{mc\_own}}$ and $\mathbf{G}_{\mathrm{cm\_own}}$ can be converted to each other, the actual output of the graph encoder is $\mathbf{G}_{\mathrm{mc\_call}}$ and $\mathbf{G}_{\mathrm{mc\_own}}$. To save space and improve efficiency, we share information of these two graphs. Therefore, only $\mathbf{V}_{\mathrm{m}}$, $\mathbf{V}_{\mathrm{c}}$, $\mathbf{G}_{\mathrm{mc\_call}}$, $\mathbf{G}_{\mathrm{mc\_own}}$, $\mathbf{F}_{\mathrm{m}}$ and $\mathbf{F}_{\mathrm{c}}$ are needed to obtain the four heterogeneous graph representations.

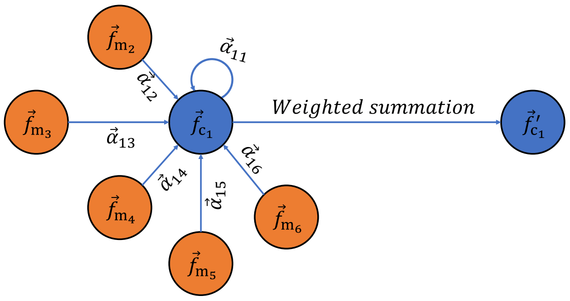


Fig. 9 An illustration of GAT, where $\vec{f}_{c_1}$, $\vec{f}_{m_2}$, $\vec{f}_{m_3}$, $\vec{f}_{m_4}$, $\vec{f}_{m_5}$ and $\vec{f}_{m_6}$ represent the node features of class c_1, methods m_2, m_3, m_4, m_5 and m_6, respectively

4.4 Symmetric feature fusion learning model

4.4.1 Graph attention network

We use the directed graph $\mathbf{G}_{\text{cm\_call}}$, representing the invocation relationships between methods and classes, to illustrate how the Graph Attention Network (GAT) (Velickovic et al. 2018) performs message passing.

As shown in Fig. 9, class c_1 is called by five methods. Since these methods may potentially have code smells, we need to let class c_1 know which methods call it, as well as the semantic and position information of these methods. Obviously, these methods cannot have the same importance, so we use an Attention Mechanism (Bahdanau et al. 2015)

$$a : \mathbb{R}^d \times \mathbb{R}^d \rightarrow \mathbb{R} \tag{39}$$

to calculate the attention coefficients between the class node and the method nodes. The attention coefficient is defined as

$$e_{1j} = a\left(\mathbf{W}\vec{f}_{c_1}, \mathbf{W}\vec{f}_{m_j}\right), \tag{40}$$

where $\mathbf{W} \in \mathbb{R}^{d\times d}$ is a trainable weight matrix. Also, we need to calculate the attention coefficient between the class node and itself

$$e_{11} = a\left(\mathbf{W}\vec{f}_{c_1}, \mathbf{W}\vec{f}_{c_1}\right). \tag{41}$$

We use softmax to normalize the attention coefficients, ensuring that the sum of all attention coefficients is 1:

$$\alpha_{1j} = \text{softmax}_j(e_{1j}) = \frac{\exp(e_{1j})}{\sum_{j=1}^{6} \exp(e_{1j})} \tag{42}$$

Furthermore, by weighted summation and activation function ELU, we can obtain the updated node representation of class c_1:

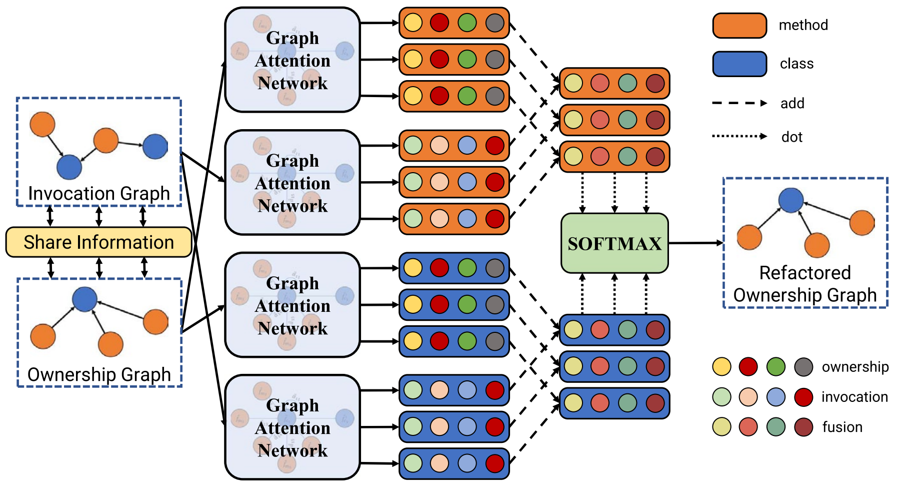


Fig. 10 Structure of symmetric feature fusion learning model

$$\vec{f}'_{c_1} = \mathrm{ELU}\left(\sum_{j=1}^{6} \alpha_{1j}\mathbf{W}\vec{f}_j\right), \tag{43}$$

where $\vec{f}_1$ is actually $\vec{f}_{c_1}$, $\vec{f}_2$ is actually $\vec{f}_{m_2}$.

To stabilize the learning process of self-attention, multi-head attention is introduced (Vaswani et al. 2017). We use K independent Attention Mechanisms and take their average as the final class node representation:

$$\vec{f}'_{c_1} = \mathrm{ELU}\left(\frac{1}{K}\sum_{k=1}^{K}\sum_{j=1}^{6} \alpha_{1j}^{k}\mathbf{W}^{k}\vec{f}_j\right). \tag{44}$$

Thus, by using GAT for message passing, the updated feature matrix of class nodes can be represented as $\mathbf{F}_{\text{cm\_call}} \in \mathbb{R}^{c\times d}$.

4.4.2 Symmetric feature fusion learning

As shown in Fig. 10, we feed two heterogeneous graphs into GATs and obtain the four feature representations:

- $\mathbf{F}_{\text{mc\_call}} \in \mathbb{R}^{m\times d}$ takes into account the semantic and position information of the methods themselves, as well as the semantic and position information of the classes they call.
- $\mathbf{F}_{\text{cm\_call}} \in \mathbb{R}^{c\times d}$ takes into account the semantic and position information of the classes themselves, as well as the semantic and position information of the methods that call them.
- $\mathbf{F}_{\text{mc\_own}} \in \mathbb{R}^{m\times d}$ takes into account the semantic and position information of the methods themselves, as well as the semantic and position information of the classes they belong to.

- $\mathbf{F}_{\mathrm{cm\_own}} \in \mathbb{R}^{c \times d}$ takes into account the semantic and position information of the classes themselves, as well as the semantic and position information of the methods they have.

We perform feature fusion to obtain

$$\mathbf{F}'_{\mathrm{m}} = \mathbf{F}_{\mathrm{mc\_call}} + \mathbf{F}_{\mathrm{mc\_own}} \in \mathbb{R}^{m \times d}, \tag{45}$$

which represents the final node embeddings for methods. Similarly, we can obtain

$$\mathbf{F}'_{\mathrm{c}} = \mathbf{F}_{\mathrm{cm\_call}} + \mathbf{F}_{\mathrm{cm\_own}} \in \mathbb{R}^{c \times d}, \tag{46}$$

which represents the final node embeddings for classes.

Thus, each method representation vector of $\mathbf{F}'_{\mathrm{m}}$ knows the information of its class, the information of classes it calls, and the information of itself. Symmetrically, each class representation vector of $\mathbf{F}'_{\mathrm{c}}$ knows the information of its methods, the information of methods that call it, and the information of itself. We call this process Symmetric Feature Fusion Learning (SFFL), which ensures that $\mathbf{F}'_{\mathrm{m}}$ and $\mathbf{F}'_{\mathrm{c}}$ share common information while also retaining their individual characteristics.

In summary, SFFL achieves a balance between shared information and individual characteristics, thereby enabling a more holistic representation of methods and classes. It enhances the capabilities of the model by leveraging the synergistic effects of combining both types of features.

4.4.3 Graph reconstruction

To obtain the reconstructed method-class ownership graph, we adopt the common practice of graph link prediction:

$$\hat{\mathbf{A}}_{\mathrm{mc\_own}} = \mathrm{softmax}(\mathbf{F}'_{\mathrm{m}} \times \mathbf{F}'^{T}_{\mathrm{c}}) \in \mathbb{R}^{m \times c}, \tag{47}$$

where softmax normalizes each row of the matrix to ensure that the probabilities of a method belonging to all classes sum up to 1. In the predicted adjacency matrix $\hat{\mathbf{A}}_{\mathrm{mc\_own}}$, the element in the i-th row and j-th column represents the probability that method i belongs to class j. The target class for method i can be expressed as

$$\mathrm{target\_class}_i = \arg\max_j \left(\hat{\mathbf{A}}_{\mathrm{mc\_own}}[i,j] \right). \tag{48}$$

Since it can be seen as a multi-classification problem, we employ the cross-entropy loss function to compute the model's loss:

$$\mathcal{L} = -\sum_{i=1}^{m} \sum_{j=1}^{c} \mathbf{A}_{\mathrm{mc\_own}}[i,j] \log(\hat{\mathbf{A}}_{\mathrm{mc\_own}}[i,j]), \tag{49}$$

where $\mathbf{A}_{\mathrm{mc\_own}}[i,j]$ represents whether the method i belongs to class j.

4.5 Detection and refactoring

Unlike SCG, the final output of this model is the adjacency matrix representing the ownership relationships between methods and classes. This matrix covers all methods and classes in the project. With Eq. (48), we can directly determine the class to which each method should belong.

For a method, if its ownership remains the same before and after prediction, it does not need to be refactored. However, if its ownership changes to two different classes before and after prediction, the method should be moved to the target class.

In fact, this approach focuses solely on the final ownership of each method, bypassing the detection phase of *feature envy*. However, it can still be inferred whether a method exhibits *feature envy* based on whether it requires refactoring or not.

5 Evaluation metrics

In previous studies, researchers (Liu et al. 2018; Wang et al. 2020; Yu et al. 2022) have addressed *feature envy* refactoring as two distinct problems: *feature envy* detection and smelly method refactoring recommendation. As a result, they employ different evaluation metrics for each problem. In this section, we review the existing evaluation metrics and propose new ones that are effective for both *feature envy* detection and refactoring.

5.1 Traditional metrics for detection and refactoring

Researchers (Liu et al. 2018; Wang et al. 2020; Yu et al. 2022; Zhang and Jia 2022; Sharma et al. 2021; Guo et al. 2019; Yin et al. 2021; Wang et al. 2020; Zhao et al. 2022; Ma et al. 2023) commonly use precision ($precision_1$), recall ($recall_1$), and F1-score ($F_1\text{-score}_1$) to evaluate the performance of *feature envy* detection. These metrics are defined as follows:

$$precision_1 = \frac{True\ Smells}{Pred\ Smells}, \tag{50}$$

$$recall_1 = \frac{True\ Smells}{Total\ Smells}, \tag{51}$$

$$F_1\text{-score}_1 = 2 \cdot \frac{precision_1 \cdot recall_1}{precision_1 + recall_1}, \tag{52}$$

where *True Smells* refers to the number of correctly detected smelly methods, *Pred Smells* refers to the number of methods predicted as smells, and *Total Smells* refers to the total number of smelly methods.

To evaluate the performance of refactoring, Liu et al. (2018) introduce the accuracy metric, defined as follows:

$$accuracy = \frac{True\ Refactoring\ Smells}{True\ Smells}, \tag{53}$$

where *True Refactoring Smells* refers to the number of correctly refactored code smells. This metric has been widely used as the sole evaluation metric for refactoring (Liu et al. 2018; Yu et al. 2022; Wang et al. 2020; Ma et al. 2023).

5.2 New metrics for detection and refactoring

We argue that the *accuracy* cannot objectively evaluate the performance of refactoring. For instance, consider two approaches: approach A detects 20 true code smells, out of which 10 are accurately refactored, while approach B detects 100 true code smells and accurately refactors 30 of them. Approach A achieves *accuracy* of 50%, while approach B only has a 30% *accuracy*. This unfair result is primarily due to the fact that the *True Smells* detected by each approach are different.

Furthermore, relying solely on *accuracy* may not provide a comprehensive and fair assessment of the refactoring effectiveness. In real-world scenarios, it is difficult to determine the correctly detected code smells (i.e., *True Smells*), so all the detected code smells (i.e., *Pred Smells*) should be refactored. Also, *Total Smells* should be taken into account as it can measure the proportion of correctly refactored smells (i.e., *True Refactoring Smells*).

As a result, we propose three new evaluation metrics that treat the correctness of refactorings as a binary classification problem:

$$precision_2 = \frac{True\ Refactoring\ Smells}{Pred\ Smells}, \tag{54}$$

$$recall_2 = \frac{True\ Refactoring\ Smells}{Total\ Smells}, \tag{55}$$

$$F_1\text{-score}_2 = 2 \cdot \frac{precision_2 \cdot recall_2}{precision_2 + recall_2}. \tag{56}$$

In fact, besides having practical meanings, these new metrics can be derived from the traditional metrics:

$$precision_2 = accuracy \cdot precision_1, \tag{57}$$

$$recall_2 = accuracy \cdot recall_1. \tag{58}$$

Since not all methods that are correctly detected as smells will be correctly refactored (i.e., *accuracy* is less than 1), $precision_2$ will not exceed $precision_1$. The relationship between $recall_2$ and $recall_1$ is also similar. We believe that these three new metrics can replace the previous ones because they provide a more comprehensive evaluation of the detection and refactoring performance.

6 Dataset and experimental settings

We use the dataset[5] collected and labeled by Sharma and Kessentini (2021), which has been frequently downloaded, as many as 14,200 times up to Aug. 2023. This dataset has also been used as the ground truth in recent studies (Nandani et al. 2023; Hadj-Kacem and Bouassida 2022). We select five projects from this dataset that meet the following criteria:

- The last update is within two years.
- The number of stars on GitHub exceeds 2,000.
- The number of methods in the project exceeds 2,000 and the number of classes exceeds 300.
- The number of methods affected by *feature envy* is among the top five.

The details of the selected projects are described in Table 2. In this dataset, Sharma and Kessentini (2021) annotated the methods with *feature envy* and provided the recommended refactoring classes for these smelly methods.

To ensure consistent and reliable experimental results while minimizing the impact of randomly selected training data, we adhere to the following procedure:

- *Random data splitting* Randomly divide the nodes of the graph into training, validation, and test sets with proportions of 60%, 20%, and 20% respectively. This ensures a suitable distribution of data for training, validation, and testing.

Table 2 Dataset description

Project	Version	Domain	NOC[1]	NOM[2]	FE[3]	SR[4]
ActiveMQ	5.16.0	Message oriented middleware	2,907	15,482	585	3.78%
Alluxio	2.4.0	Distributed storage system	1,501	7,019	367	5.23%
BinNavi	6.1.0	Binary nalysis IDE	3,331	10,852	996	9.18%
Kafka	2.5.0	Distributed event flow platform	1,819	9,818	446	4.54%
Realm-java	10.2.0	Mobile database	372	2,466	260	10.54%

[1] Number of classes

[2] Number of methods

[3] Number of methods with feature envy

[4] Smell ratio = FE / NOM

[5] https://doi.org/10.5281/zenodo.4468361

Table 3 Optimal parameter settings for SCG and SFFL

Parameter	Approach	
	SCG	SFFL
Dimension of word embedding	–	256
Dropout rate	0.1	
Learning rate	1E-3	
Number of attention heads for GAT	–	8
Training epochs of model	1000	2000
Training epochs of Word2Vec	–	300
Weight decay	5E-4	
λ	1E-6	–

- *Fixed random seed* Set a fixed random seed chosen from the set $\{1, 2, 3, 4, 5\}$ for data splitting. This ensures consistent dataset divisions across different runs, keeping the training, validation, and test sets the same for each experiment.
- *Experiment repetition* Repeat the experiments five times, each with a different random seed. This accounts for potential variability caused by random division and allows us to evaluate the model's performance on different subsets of the data.
- *Evaluation metrics averaging* Obtain evaluation metrics from each experiment and average them. By averaging the results, we mitigate the impact of random divisions and obtain more reliable and representative performance evaluation results.

We conducted all experiments on a 64-bit computer with an Intel i9-10850K CPU and an Nvidia RTX 3060 GPU. We used AdamOptimizer (Kingma and Ba 2015) to train our model. The optimal model parameters are shown in Table 3.

7 The evaluation of SCG

In this section, we evaluate the proposed approach SCG on five open-source projects with *feature envy*.

7.1 Research questions

The evaluation investigates the following Research Questions (RQs):

- **RQ-SCG-1**: How does the selection of λ affect SCG performance?
- **RQ-SCG-2**: Is SCG superior to the compared ones on *feature envy* detection?
- **RQ-SCG-3**: How does SCG perform on refactoring recommendations?
- **RQ-SCG-4**: How does the graph augmenter impact the performance of SCG?
- **RQ-SCG-5**: How effective is SCG in cross-project scenarios?

RQ-SCG-1 investigates the appropriate parameter settings for better performance on *feature envy* detection. As shown in Eq. (24), the objective function comprehensively considers the loss functions of the graph augmentation and the GNN classifier, using λ to adjust the weight of both. Consequently, different parameter settings may affect the classification results of the model.

RQ-SCG-2 considers how SCG performs compared with others. To answer this question, we compare it against *JDeodorant* (Tsantalis and Chatzigeorgiou 2009), *JMove* (Terra et al. 2018) and Liu's work (Liu et al. 2018). The main reason for choosing them is that they have been widely employed as competitors (Wang et al. 2020; Yin et al. 2021; Zhao et al. 2022). In addition, they both address *feature envy* detection and refactoring, while others (Guo et al. 2019; Sharma et al. 2021; Yin et al. 2021; Zhang and Jia 2022; Zhao et al. 2022) do not consider refactoring.

RQ-SCG-3 considers how accurate SCG is in refactoring recommendations. We investigate this question because the ultimate goal of code smells detection is not to find out code smells, but to eliminate these smells through refactoring. In other words, it is significant to suggest correctly to which class the misplaced method with *feature envy* should be moved.

RQ-SCG-4 investigates how the graph augmenter contributes to the overall performance. To answer this question, we conduct the ablation experiment to observe the performance of SCG if not applying the graph augmenter.

RQ-SCG-5 investigates whether SCG still has remarkable performance in cross-project scenarios. To this end, we train the model on one project and then test it on the other four projects. In order to fit the actual application scenario, we also introduce fine-tuning part. That is, the model trained on Project A, adjusts its parameters with a small amount of data (such as 1% or 10%) from Project B, and then tests the remaining data of Project B.

7.2 RQ-SCG-1: impact of λ

In this RQ, we adjust λ to compare the performance of SCG, and determine the best or approximate best λ to achieve a better classification effect.

We empirically choose the candidate value of λ from the set $\{1\mathrm{E}-7, 5\mathrm{E}-7, 1\mathrm{E}-6, 5\mathrm{E}-6, 1\mathrm{E}-5, 5\mathrm{E}-5, 1\mathrm{E}-4\}$, and use F_1-score$_1$ to comprehensively evaluate the detection performance. When other parameters remain unchanged, we apply the model to the dataset. Figure 11 shows the impact of different λ on the performance.

RQ-SCG-1 Summary

When the value of λ comes to $1\mathrm{E}-6$, SCG achieves the best F_1-score$_1$ in *feature envy* detection. Therefore, in subsequent experiments, we set $1\mathrm{E}-6$ as the default value for λ.

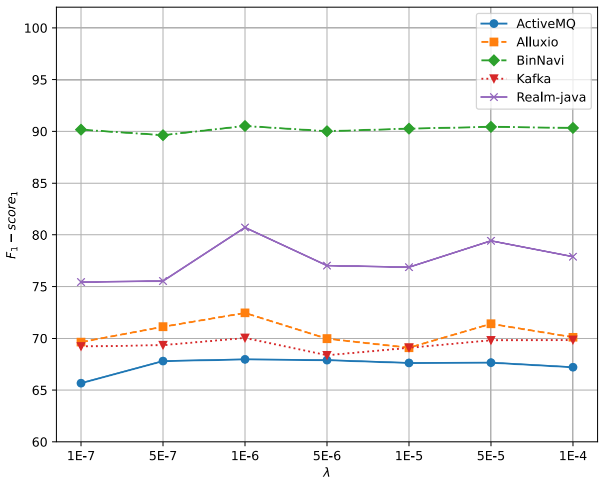


Fig. 11 Impact of different λ on the performance

7.3 RQ-SCG-2: detection of feature envy

We compare SCG with three competitors: Liu's work (Liu et al.. 2018), *JDeodorant* (Tsantalis and Chatzigeorgiou 2009) and *JMove* (Terra et al.. 2018). The basic principles of the three are introduced in Sect. 2. We keep the parameter settings of Liu's work,[6] because they have proved its rationality through a large number of experiments. *JDeodorant*[7] and *JMove*[8] are implemented as Eclipse[9] plug-ins, which do not require any specific parameters to be set.

Table 4 shows the detailed results of SCG and three competitors on five projects, from which we make the following observations:

- First, SCG significantly outperforms the three competitors on the average performance of the three metrics. Its average F_1-score$_1$ is 76.34% whereas the average F_1-score$_1$ of Liu's work, *JDeodorant* and *JMove* are 40.92%, 25.01% and 19.47%, respectively.
- Second, SCG can identify most of the *feature envy*. Compared with Liu's work, *JDeodorant* and *JMove*, it improves *recall*$_1$ from 73.13%, 18.17% and 12.72% to 79.68%, respectively.
- Third, the *precision*$_1$ (73.62%) of SCG is dramatically greater than those (28.55%, 40.70% and 44.38%) of Liu's work, *JDeodorant* and *JMove*, respectively.

[6] https://github.com/liuhuigmail/FeatureEnvy

[7] https://github.com/tsantalis/JDeodorant

[8] https://github.com/aserg-ufmg/jmove

[9] https://www.eclipse.org/

Table 4 Evaluation results on feature envy detection

Project	SCG			Liu's work			JDeodorant			JMove		
	$precision_1$(%)	$recall_1$(%)	F_1-score$_1$(%)	$precision_1$(%)	$recall_1$(%)	F_1-score$_1$(%)	$precision_1$(%)	$recall_1$(%)	F_1-score$_1$(%)	$precision_1$(%)	$recall_1$(%)	F_1-score$_1$(%)
ActiveMQ	**63.97**	**72.65**	**67.97**	35.64	80.39	49.38	60.93	27.82	38.20	63.16	19.64	29.96
Alluxio	**67.32**	**78.65**	**72.46**	22.46	64.10	33.26	61.94	24.34	34.95	50.96	15.54	23.82
BinNavi	**93.07**	**88.20**	**90.52**	25.35	78.92	38.37	17.33	7.04	10.01	27.78	9.05	13.66
Kafka	**67.30**	**73.11**	**70.03**	28.06	73.17	40.57	30.29	17.89	22.50	32.92	12.99	18.63
Realm-java	**76.42**	**85.77**	**80.71**	31.22	69.06	43.00	33.03	13.74	19.41	47.06	6.40	11.27
Average	**73.62**	**79.68**	**76.34**	28.55	73.13	40.92	40.70	18.17	25.01	44.38	12.72	19.47

The best performance metric for a specific project is in bold

Table 5 Accuracy in refactoring recommendation

Project	SCG (%)	Liu's work (%)	JDeodorant (%)	JMove (%)
ActiveMQ	**97.60**	63.45	60.12	59.82
Alluxio	**77.76**	55.88	62.94	65.88
BinNavi	**96.27**	21.64	29.76	12.77
Kafka	**88.01**	62.07	62.04	65.45
Realm-java	**40.84**	53.68	66.67	48.28
Average	**80.10**	51.34	56.31	50.44

The best performance metric for a specific project is in bold

The reason why SCG performs better is that it can effectively capture the calling relationships between methods by leveraging GNN. As for efficiency, for each project, SCG took about 3 min in model training while Liu's work took about 16 min. Since *JDeodorant* and *JMove* are both Eclipse plug-ins, they do not need training. On the other hand, SCG took less than 2 s in testing, while Liu's work took 5 min and both *JDeodorant* and *JMove* took more than 20 min.

RQ-SCG-2 Summary

SCG demonstrates significant improvements over three competitors in terms of both time efficiency and performance in detecting *feature envy*.

7.4 RQ-SCG-3: refactoring recommendations

Methods affected by *feature envy* should be relocated and moved to the appropriate target classes. However, when these methods have multiple target classes at the same time (i.e., when the method is more interested in multiple external classes than its own class), it is necessary to recommend the best one.

We still compare SCG with the approaches provided by Liu's work (Liu et al. 2018), *JDeodorant* (Tsantalis and Chatzigeorgiou 2009) and *JMove* (Terra et al. 2018), and take the *accuracy* as the evaluation metric. Table 5 lists the performance comparison of SCG and the three competitors in *accuracy*.

As Table 5 indicates, firstly, SCG works well when recommending target classes for smelly methods. Its *accuracy* ranges from 40.84% to 97.60%, and the average *accuracy* is as high as 80.10%. In other words, for more than four fifths of the methods detected with *feature envy*, SCG can provide the correct target classes. In addition, SCG has higher *accuracy* than three competitors in refactoring recommendations. Specifically, compared with Liu's work, *JDeodorant* and *JMove*, it improves average *accuracy* from 51.34%, 56.31% and 50.44% to 80.10%, respectively.

Table 6 Performance comparison with and without GraphSMOTE

Project	With GraphSMOTE			Without GraphSMOTE		
	$precision_1$(%)	$recall_1$(%)	F_1-score$_1$(%)	$precision_1$(%)	$recall_1$(%)	F_1-score$_1$(%)
ActiveMQ	63.97	**72.65**	**67.97**	**81.17**	57.27	66.94
Alluxio	67.32	**78.65**	**72.46**	**80.06**	62.70	70.17
BinNavi	93.07	**88.20**	90.52	**96.00**	86.10	**90.77**
Kafka	67.30	**73.11**	**70.03**	**81.36**	56.00	66.11
Realm-java	76.42	**85.77**	**80.71**	**79.99**	74.62	77.04
Average	73.62	**79.68**	**76.34**	**83.71**	67.34	74.21

The best performance metric for a specific project is in bold

RQ-SCG-3 Summary

For the methods affected by *feature envy*, our approach SCG has better performance than three competitors in refactoring recommendations.

7.5 RQ-SCG-4: ablation experiment

In most projects, the number of smelly methods is much less than that of normal methods. Under such a circumstance, training the classifier with unbalanced data tends to underestimate the smelly methods. To address such a problem, in SCG, we exploit GraphSMOTE (Zhao et al. 2021) as the graph augmenter to generate the smelly nodes and edges connected to these nodes.

As Table 6 indicates, GraphSMOTE sacrifices some $precision_1$ in exchange for improved $recall_1$, resulting in an average F_1-score$_1$ improvement of 2.13% (=76.34%−74.21%). Compared to not using GraphSMOTE, the utilization of GraphSMOTE leads to a more balanced performance in the model, with average $precision_1$ and $recall_1$ both maintained at over 70%.

RQ-SCG-4 Summary

The comprehensive performance of SCG can be improved if GraphSMOTE is introduced.

7.6 RQ-SCG-5: cross-project scenarios

In this RQ, we evaluate SCG in cross-project scenarios and set the F_1-score$_1$ as a comprehensive evaluation indicator.

First, we simply employ the model trained on one project to detect the smells of another project. Second, we input a small amount of test data into the model in advance to slightly adjust its internal parameters. The main reason is that, in the actual application scenario, the inflexible model parameters can not effectively detect smells of the target project (Bui et al. 2021), while a small amount of

Table 7 The F_1-score$_1$ of SCG in cross-project scenarios

Tested on	Fine-tune	Trained on				
		ActiveMQ (%)	Alluxio (%)	BinNavi (%)	Kafka (%)	Realm-java (%)
ActiveMQ	None	–	46.20	**49.65**	43.11	39.45
	1%	–	63.04	**63.11**	60.70	62.68
	5%	–	**68.03**	65.18	65.51	64.29
	10%	–	**65.80**	64.42	65.26	64.31
Alluxio	None	42.37	–	71.37	**71.89**	48.50
	1%	58.68	–	62.52	**64.36**	58.91
	5%	65.45	–	62.83	65.25	**66.18**
	10%	**70.81**	–	69.63	68.50	67.32
BinNavi	None	71.88	81.74	–	**85.08**	75.93
	1%	**90.17**	89.50	–	88.83	85.78
	5%	**88.92**	88.73	–	85.56	86.99
	10%	90.75	**91.06**	–	89.96	87.56
Kafka	None	49.44	64.83	**66.28**	–	49.03
	1%	45.52	**56.66**	56.07	–	56.16
	5%	67.27	**67.97**	65.58	–	66.15
	10%	**68.56**	66.97	67.62	–	65.29
Realm-java	None	13.33	29.11	**32.95**	31.35	–
	1%	63.94	65.46	**65.60**	63.66	–
	5%	75.34	70.94	**76.73**	73.36	–
	10%	**77.51**	69.11	76.48	74.52	–
Average	None	44.26	55.47	55.06	**57.86**	53.23
	1%	64.58	68.66	61.82	**69.39**	65.88
	5%	**74.25**	73.92	67.58	72.42	70.90
	10%	**76.91**	73.23	69.54	74.56	71.12

With specific fine-tuning data, the best F_1-score$_1$ in cross-project scenarios is in bold

manually labeled data (such as 1% or 10%) can improve the generalization of the model. Different from training from scratch, the epoch rounds of the fine-tuning experiment are set as 400.

Table 7 shows the F_1-score$_1$ of SCG in cross-project scenarios, where **None** refers to not using any data for fine-tuning, **1%**, **5%** and **10%** indicate the use of 1%, 5% and 10% of the data for fine-tuning, respectively. For example, the model trained on BinNavi achieves an F_1-score$_1$ of 49.65% on ActiveMQ without fine-tuning. However, when the same model trained on BinNavi is fine-tuned with only 1% of the data from ActiveMQ, it achieves a significantly higher F_1-score$_1$ of 63.11% on the remaining 99% of the data. Of course, if fine-tuned on more data, such as 5% or 10% of the target project, the performance will be improved further. Note that, only with 1% data of target project, the model trained on Kafka can easily obtain 69.39% in terms of average F_1-score$_1$, which is close to the average F_1-score$_1$ in within-project scenarios .

RQ-SCG-5 Summary

SCG still has remarkable performance in cross-project scenarios, and can easily obtain a similar performance compared with within-project scenarios only with a little labeled data.

8 The evaluation of SFFL

In this section, we evaluate the proposed approach SFFL on five open-source projects with *feature envy*.

8.1 Research questions

The evaluation investigates the following Research Questions (RQs):

- *RQ-SFFL-1* How do the number of training epochs and the vector size in Word2Vec affect the performance of SFFL?
- *RQ-SFFL-2* How do positional and semantic embeddings impact the performance of SFFL?
- *RQ-SFFL-3* What are the advantages of using a Graph Attention Network (GAT) compared to other graph convolution techniques?
- *RQ-SFFL-4* How does SFFL perform in *feature envy* refactoring compared with SCG?
- *RQ-SFFL-5* How effective is SFFL in cross-project scenarios?

RQ-SFFL-1 examines the impact of parameter settings in Word2Vec (Mikolov et al. 2013) on model performance. The number of training epochs and vector size can affect the quality of word embeddings. High epochs may result in overfitting, while low epochs may lead to lower-quality embeddings. Similarly, a large vector size increases complexity and training time, while a small vector size may not capture semantic information effectively. Thus, experiments are needed to determine the optimal parameter settings.

RQ-SFFL-2 investigates the impact of positional and semantic embeddings on the model. We will explore three strategies: 1) using only positional embeddings, 2) using only semantic embeddings, and 3) using both positional and semantic embeddings. Positional embeddings provide information about node positions in the graph, aiding in capturing local and global information. Semantic embeddings capture distinctive features of nodes, enabling the model to understand semantic similarities between methods and classes. By evaluating the model's performance with these different strategies, we can gain insights into the influence of positional and semantic embeddings.

RQ-SFFL-3 focuses on exploring the advantages of using a Graph Attention Network (GAT) (Velickovic et al. 2018) in SFFL. For comparison, we consider two other graph convolution techniques: GCN (Kipf and Welling 2017) and GraphSAGE (Hamilton et al. 2017). GCN updates the node feature by averaging the features of its

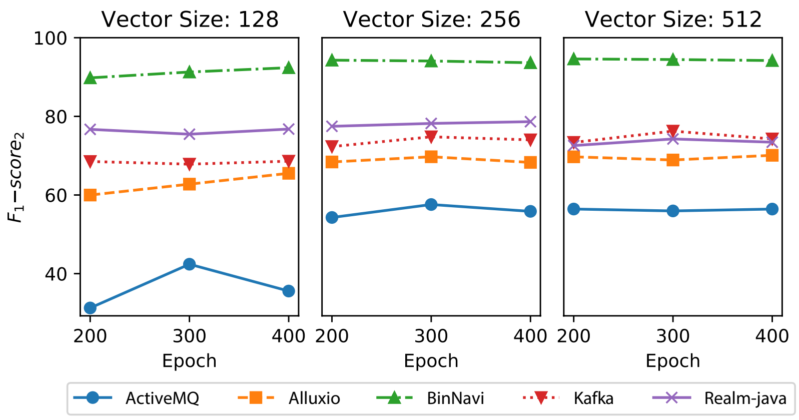


Fig. 12 Comparison on varying vector sizes and epochs

neighboring nodes. GraphSAGE uses specific aggregation functions to aggregate the features of neighboring nodes and appends the aggregated results to the current node for updating.

RQ-SFFL-4 considers how SFFL performs in *feature envy* refactoring. Considering that SCG demonstrates significantly superior performance in terms of detection and refactoring compared to Liu's work (Liu et al. 2018), *JDeodorant* (Tsantalis and Chatzigeorgiou 2009), and *Jmove* (Terra et al. 2018), we consider it as the competitor for comparison. The specific methodologies of these approaches are presented in Sect. 2.

RQ-SFFL-5 investigates whether SFFL still has remarkable performance in cross-project scenarios. To this end, we train the model on one project and then test it on the other four projects. In order to fit the actual application scenario, we also introduce fine-tuning part. That is, the model trained on Project A, adjusts its parameters with a small amount of data (such as 1% or 10%) from Project B, and then tests the remaining data of Project B.

8.2 RQ-SFFL-1: parameters for Word2Vec

Due to the impact of the number of training epochs and word vector size on model performance, we conducted experiments on five projects, calculating F_1-score$_2$ with different configurations. For the number of training epochs, we test values of 300, 400 and 500. Regarding the word vector size, we experiment with sizes of 128, 256 and 512.

Figure 12 presents the experimental results. We observe that when the vector size is set to 128, the F_1-score$_2$ for each project is lower than that of vector sizes 256 and 512. However, when the vector size increases from 256 to 512, the model's performance does not significantly improve, and the F_1-score$_2$ of Realm-java even decreases. When the vector size is fixed at 256 or 512, we find that an epoch number of 300 yields balanced results, as SFFL achieves the best performance on most projects at this epoch.

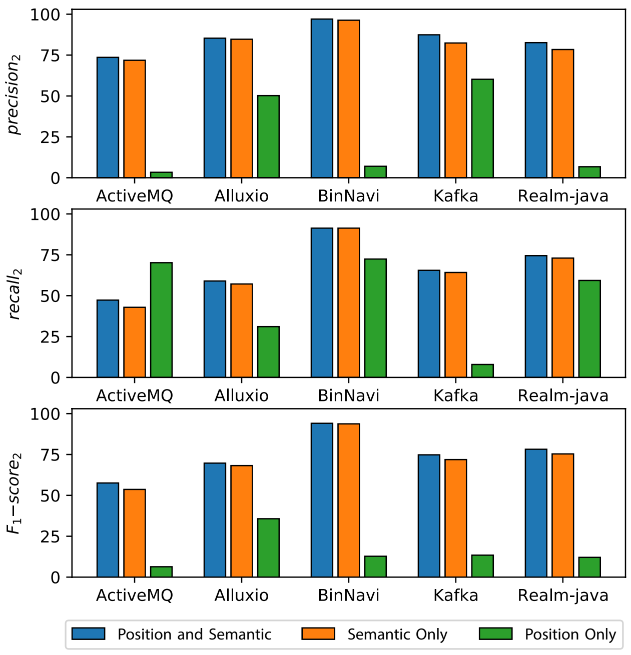


Fig. 13 The results of three information input strategies

RQ-SFFL-1 Summary

A vector size of 256 and an epoch number of 300 achieve a good balance between performance and efficiency. These settings are utilized for all subsequent experiments.

8.3 RQ-SFFL-2: impact of semantic and position information

In order to understand the contribution of semantic and position information to our model, we explore three strategies: 1) using only positional information, 2) using only semantic information, and 3) using both of them. To comprehensively evaluate the impact of these strategies on model performance, we calculate $precision_2$, $recall_2$ and F_1-score$_2$.

Figure 13 shows that using both positional and semantic information is the optimal strategy, while relying solely on positional information results in the poorest performance. An interesting finding is that when using only positional information, the model tends to be overfit. Specifically, when $precision_2$ is high, $recall_2$ is extremely low, and vice versa. The main reason is that when using solely positional information, the model can only understand the relative positions of nodes in the graph without considering their individual characteristics. The inclusion of semantic information effectively addresses this issue.

RQ-SFFL-2 Summary

Semantic and positional information complement each other, and using both simultaneously enables the model to achieve optimal performance.

8.4 RQ-SFFL-3: advantage of GAT

To investigate whether GAT (Velickovic et al. 2018) can effectively capture the invocation and ownership information between methods and classes, we introduce GCN (Kipf and Welling 2017) and GraphSAGE (Hamilton et al 2017) for comparison.

As shown in Table 8, GAT slightly outperforms GCN and GraphSAGE in terms of the average values of the three evaluation metrics. We believe that the small difference is due to the adoption of Symmetric Feature Fusion Learning (SFFL), which ensures that the neighboring node features learned through graph convolution are shared between method nodes and class nodes, while still preserving their unique features. Therefore, any graph convolution technique can achieve good performance with SFFL.

RQ-SFFL-3 Summary

GAT exhibits a slight advantage over GCN and GraphSAGE in the three evaluation metrics. The small difference is attributed to the adoption of Symmetric Feature Fusion Learning, which enables any graph convolution technique to perform well.

8.5 RQ-SFFL-4: performance of SFFL

To investigate how SFFL performs in *feature envy* detection and refactoring, we compare it with SCG on traditional and new evaluation metrics. The definitions of these metrics can be found in Sect. 5.

As Table 9 indicates, SCG and SFFL have alternating victories in *feature envy* detection shown by $recall_1$ and F_1-score$_1$. However, when we pay attention to refactoring, the superiority of SFFL becomes evident. Especially in the case of Realme-java, SFFL achieves an F_1-score$_2$ that exceeds SCG by an absolute value of 45.16% (=78.20%−33.04%).

In the case of the ActiveMQ, it is observed that SFFL does not outperform SCG. The main reason is that ActiveMQ has the highest number of methods and classes, but the lowest proportion of *feature envy*, as shown in Table 2. SFFL does not have the capability to use data augmentation techniques (we discuss the reason in Sect. 9.4), while SCG utilizes graph augmentation to generate additional positive samples for training. Therefore, in this project with a highly imbalanced distribution of positive and negative samples, SCG performs better than SFFL.

Furthermore, the experimental results also indicate that *accuracy* does not adequately reflect the performance of refactoring. For example, in the case of Kafka,

Table 8 Evaluation results on different graph convolution techniques

Project	GAT			GCN			GraphSAGE		
	$precision_2$(%)	$recall_2$(%)	F_1-score$_2$(%)	$precision_2$(%)	$recall_2$(%)	F_1-score$_2$(%)	$precision_2$(%)	$recall_2$(%)	F_1-score$_2$(%)
ActiveMQ	73.61	**47.29**	**57.58**	**77.82**	41.03	53.64	74.28	44.41	55.49
Alluxio	85.33	**58.98**	**69.72**	**87.12**	57.40	69.16	84.66	57.12	68.13
BinNavi	**97.03**	91.30	**94.08**	95.61	91.30	93.39	96.33	**91.60**	93.90
Kafka	**87.42**	65.50	**74.79**	84.66	65.28	73.40	85.65	**65.72**	74.13
Realm-java	**82.62**	**74.47**	**78.20**	80.47	74.09	76.96	80.35	73.33	76.54
Average	**85.20**	**67.51**	**74.87**	85.13	65.82	73.31	84.25	66.44	73.64

The best performance metric for a specific project is in bold

Table 9 Evaluation results on feature envy detection and refactoring

Project	Approach	Traditional metrics				New metrics		
		$precision_1$(%)	$recall_1$(%)	F_1-score$_1$(%)	$accuracy$(%)	$precision_2$(%)	$recall_2$(%)	F_1-score$_2$(%)
ActiveMQ	SCG	63.97	**72.65**	**67.97**	**97.60**	62.48	**70.94**	**66.38**
	SFFL	**75.46**	48.48	59.03	97.56	**73.61**	47.29	57.58
Alluxio	SCG	67.32	**78.65**	**72.46**	77.76	52.27	**61.08**	56.27
	SFFL	**86.85**	60.05	70.98	**98.25**	**85.33**	58.98	**69.72**
BinNavi	SCG	93.07	88.20	90.52	96.27	89.60	84.90	87.14
	SFFL	**99.15**	**93.30**	**96.13**	**97.86**	**97.03**	**91.30**	**94.08**
Kafka	SCG	67.30	**73.11**	70.03	88.01	59.18	64.22	61.55
	SFFL	**88.88**	66.60	**76.05**	**98.39**	**87.42**	**65.50**	**74.79**
Realm-java	SCG	76.42	**85.77**	80.71	40.84	31.36	35.00	33.04
	SFFL	**88.29**	79.53	**83.53**	**93.61**	**82.62**	**74.47**	**78.20**
Average	SCG	73.62	**79.68**	76.34	80.10	58.98	63.23	60.88
	SFFL	**87.73**	69.59	**77.14**	**97.13**	**85.20**	**67.51**	**74.87**

The best performance metric for a specific project is in bold

Table 10 The F_1-score$_2$ of SFFL in cross-project scenarios

Tested on	Fine-tune	Trained on				
		ActiveMQ (%)	Alluxio (%)	BinNavi (%)	Kafka (%)	Realm-java (%)
ActiveMQ	None	–	0.00	**0.20**	0.05	0.04
	1%	–	3.96	**4.77**	4.16	3.13
	5%	–	**5.53**	4.00	2.68	2.21
	10%	–	**10.10**	9.09	9.37	8.81
Alluxio	None	0.00	–	2.12	2.96	**10.96**
	1%	9.68	–	7.80	22.59	**26.18**
	5%	41.48	–	39.31	**42.12**	40.82
	10%	43.95	–	50.84	**51.57**	43.09
BinNavi	None	0.30	**0.99**	–	0.22	0.04
	1%	42.93	43.42	–	**52.11**	46.47
	5%	**88.04**	81.19	–	82.43	81.03
	10%	**92.38**	86.23	–	90.39	88.37
Kafka	None	0.00	0.00	0.00	–	0.00
	1%	**8.55**	2.28	5.28	–	3.45
	5%	**60.43**	57.77	59.78	–	46.38
	10%	64.08	64.17	61.76	–	**64.23**
Realm-java	None	5.71	**11.16**	1.30	4.12	–
	1%	30.01	23.11	**32.73**	25.79	–
	5%	**58.75**	52.20	56.11	57.88	–
	10%	66.67	65.97	68.19	**68.22**	–
Average	None	1.50	**3.04**	0.91	1.84	2.76
	1%	22.79	18.19	12.64	**26.16**	19.81
	5%	**62.18**	49.17	39.80	46.28	42.61
	10%	**66.77**	56.62	47.47	54.89	51.12

With specific fine-tuning data, the best F_1-score$_2$ in cross-project scenarios is in bold

although SCG achieves an *accuracy* as high as 88.01%, it actually correctly refactors only 59 out of 100 methods predicted to require refactoring.

RQ-SFFL-4 Summary

SFFL achieves excellent performance in both *feature envy* detection and refactoring, particularly in refactoring where the average F_1-score$_2$ surpasses SCG by 13.99% (= 74.87–60.88%).

8.6 RQ-SFFL-5: cross-project scenarios

In this RQ, we evaluate SFFL in cross-project scenarios and set the F_1-score$_2$ as a comprehensive evaluation indicator. We train the model on one project and employ it to refactor all methods in another project. We take a similar setup as described in Sect. 7.6, with four strategies: no fine-tuning, fine-tuning with 1%, 5% and 10% data of the target project. Additionally, we set the number of fine-tuning epochs to 400 and use the same random seed to ensure that the fine-tuning is performed with the same data as SCG.

As shown in Table 10, without fine-tuning, SFFL is unable to refactor any smelly methods in cross-project scenarios (we discuss it in Sect. 9.5). When fine-tuning with 1% of the data, SFFL shows some improvement in refactoring performance. When the fine-tuning data is increased to 5%, the model trained on ActiveMQ achieves an average F_1-score$_2$ of 62.18% on the other four projects, which is 12.69% (=74.87%−62.18%) lower compared to that of within-project scenarios.

RQ-SFFL-5 Summary

When fine-tuning with 5% of the data, SFFL achieves good refactoring performance in cross-project scenarios.

9 Discussion

9.1 Dataset

In this study, we chose not to use the dataset provided by Liu et al. (2018), despite its widespread usage in related studies (Wang et al. 2020; Yin et al. 2021; Zhao et al. 2022). There are three main reasons for this decision.

Firstly, the dataset is artificially constructed based on the assumption that there is no *feature envy* in the project. It artificially moves methods to external classes in order to create positive samples: samples with the format

$$< \text{name}_m, \text{name}_{ec}, \text{name}_{tc}, \text{dist}(m, ec), \text{dist}(m, tc) > \tag{59}$$

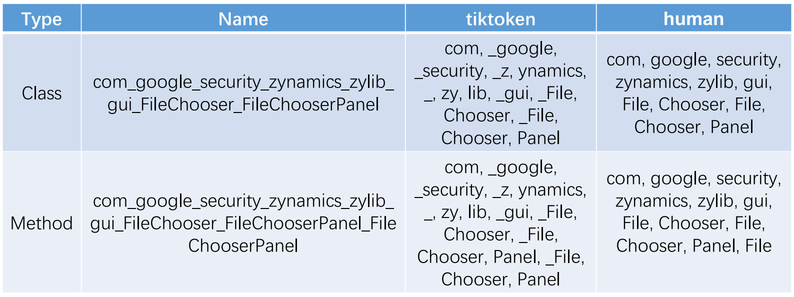
are labeled as negative, while samples with the format

$$< \text{name}_m, \text{name}_{tc}, \text{name}_{ec}, \text{dist}(m, tc), \text{dist}(m, ec) > \tag{60}$$

are labeled as positive. Furthermore, the dataset artificially sets the ratio of positive samples to 55% when generating samples. In reality, however, the ratio of positive samples is much lower than 55%. Therefore, this dataset may not accurately reflect the presence of *feature envy* in real-world projects.

Secondly, the distance provided in the dataset is not the raw information (method invocations and attribute accesses). Instead, it is calculated based on manually designed expressions using the raw information, as shown in Eqs. (2), (3). As a result, the distance information can not accurately represent the degree of envy a method has for a class.

Thirdly, in the process of creating the dataset, they narrow down the target classes using Eclipse JDT. However, in reality, the potential target classes for a method should be the set of all classes it calls.



Type	Name	tiktoken	human
Class	com_google_security_zynamics_zylib_gui_FileChooser_FileChooserPanel	com, _google, _security, _z, ynamics, _, zy, lib, _gui, _File, Chooser, _File, Chooser, Panel	com, google, security, zynamics, zylib, gui, File, Chooser, File, Chooser, Panel
Method	com_google_security_zynamics_zylib_gui_FileChooser_FileChooserPanel_FileChooserPanel	com, _google, _security, _z, ynamics, _, zy, lib, _gui, _File, Chooser, _File, Chooser, Panel, _File, Chooser, Panel	com, google, security, zynamics, zylib, gui, File, Chooser, File, Chooser, Panel, File

Fig. 14 Examples of different tokenization results

9.2 Tokenization

In this study, we utilize *tiktoken* as the tokenizer, which produces tokenization results that may not be easily interpretable. An alternative tokenization approach that provides more intuitive results is to split the tokens based on underscores and camel case. As shown in Fig. 14, the tokenization results using underscores and camel case are more easily understandable for humans. However, our experiments have shown that using this tokenization approach results in approximately 20% lower F_1-score$_2$ compared to using *tiktoken*. The reason behind this difference may be that the tokenization results from *tiktoken* are more easily understood by the computer, as the strong language understanding capabilities of ChatGPT[10] are built on top of *tiktoken*.

9.3 Word embedding

In this study, we employ Word2Vec (Mikolov et al. 2013) to obtain semantic embeddings for each method or class name. However, in the field of natural language processing, researchers often utilize pre-trained models such as BERT (Devlin et al. 2019) to compute word embeddings for texts. We also attempted to make use of the pre-trained model. Specifically, we used the smallest-scale pre-trained BERT model, BERT-base-cased, which outputs word embedding vectors of size 768. We experimented with using the raw vectors as well as reducing their size to 256 or 512. Before encoding, BERT inserts special tokens, [CLS] and [SEP], at the beginning and end of a token sequence of length n. The final encoding result consists of $n + 2$ embedding vectors, each corresponding to a token. The vector corresponding to [CLS] is often directly utilized for classification tasks because it represents the overall embedding of the sequence. Therefore, we adopted three different strategies to obtain the semantic embedding: 1) average pooling of the vectors corresponding to all tokens; 2) directly using the vector of the token [CLS] as the semantic

[10] https://chat.openai.com/

embedding; and 3) average pooling of all vectors after removing [CLS] and [SEP]. However, the resulting F_1-score$_2$ is less than 10%.

The main reason for the low performance could be the significant differences between the target scenario of BERT and the actual scenario in this paper. BERT is primarily designed to handle longer texts, with a maximum input length of 512 tokens. The number of tokens in a method or class name, however, is typically around 13. Additionally, the 768-dimensional word vectors in BERT are designed to capture complex semantic representations, whereas the semantic complexity of method and class names within a project is often lower. Last but not least, as shown in Fig. 14, the method and class names we use are not logically ordered phrases but rather their simple combinations. BERT, however, is more suitable for capturing the semantic representation of complete sentences. These factors contribute to the suboptimal performance when using BERT.

Compared to using Word2Vec, another more intuitive method is to directly generate semantic embeddings using Doc2Vec (Le and Mikolov 2014). As the extension of Word2Vec, Doc2Vec provides a more comprehensive approach to capturing semantic information in text and caters to a wider range of natural language processing tasks. The experimental results in Sect. 8.2 indicate that a vector dimension of 256 and 300 training epochs form a more optimal configuration. Therefore, we also trained Doc2Vec using this setup. The experimental results showed that, compared to using Word2Vec, using Doc2Vec led to a 4% decrease in average F_1-score$_2$, and the training time increased by a factor of 3.44.

Similar to the reason for the poor performance of BERT, the poor performance of Doc2Vec is attributed to the fact that method names and class names are not complete sentences but rather combinations of multiple phrases. Therefore, treating them as sentences to obtain their semantic embeddings would result in poorer detection and refactoring performance. Additionally, the model of SFFL needs to capture semantic similarity between method and class names, and it tends to focus more on fine-grained semantic information (obtained from Word2Vec) rather than overall semantic information (obtained from [SEP] of BERT or Doc2Vec).

9.4 Data augmentation

Data augmentation has achieved good results in SCG, but we do not use data augmentation in SFFL. The main reason is that we cannot determine the ownership relationships for the generated smelly methods. In SCG, GraphSMOTE (Zhao et al. 2021) is used to enhance the method invocation graph, and an edge predictor is utilized to predict the existence of invocation relationships between the generated methods and the original methods. However, if GraphSMOTE is employed in SFFL, we not only need to predict the invocation relationships between the generated methods and the original methods, but also need to predict the ownership relationships between the generated methods and the original classes. The latter is precisely the actual output of SFFL, which aims to build an edge predictor to predict the ownership relationships between methods and classes. Therefore, we cannot use data augmentation techniques in SFFL.

9.5 Cross-project

In cross-project scenarios without fine-tuning, SCG still performs well, while SFFL loses its ability in both detection and refactoring, as shown in Tables 7 and 10.

The poor performance of SFFL can be attributed to the semantic differences observed across various projects. SFFL distinguishes different methods and classes using word embeddings provided by Word2Vec (Mikolov et al. 2013). However, due to semantic differences, the word embeddings trained on different projects are not interchangeable. This is a common challenge in natural language processing, where models trained on one corpus often exhibit inferior performance when applied to another corpus. Therefore, we adopt the common practice of fine-tuning with a small amount of data to enhance the model's generalization ability.

The reason why SCG can still achieve good results is that it relies solely on metric information. SCG distinguishes different methods through structural and invocation metrics, capturing their relationships for detection and refactoring. Unlike semantic information, metric information is more universal and consistent across different projects.

9.6 SCG vs SFFL

SCG and SFFL view *feature envy* detection and refactoring from two different perspectives. SCG primarily focuses on *feature envy* detection and moves smelly methods to appropriate classes based on the *calling strength*. On the other hand, SFFL aims to refactor *feature envy* and determines if a method is smelly based on whether its class ownership changes before and after refactoring.

Table 9 shows that SFFL has a significant advantage over SCG in three comprehensive metrics (i.e., $precision_2$, $recall_2$ and F_1-score$_2$) across five projects. However, SCG still has its merits. SCG outperforms SFFL in three detection-only metrics (i.e., $precision_1$, $recall_1$ and F_1-score$_1$) across five projects six times, while SFFL outperforms SCG nine times. This indicates that SFFL cannot completely replace SCG in *feature envy* detection. Moreover, SCG outperforms SFFL in scenarios with highly imbalanced data, such as in ActiveMQ. This is attributed to the capability of SCG to generate positive samples using GraphSMOTE, whereas SFFL is unable to leverage data augmentation techniques as discussed in Sect. 9.4.

Figure 15 illustrates the performance of SCG and SFFL in cross-project scenarios. When fine-tuning is not introduced, SCG significantly outperforms SFFL in both detecting and refactoring (we discuss it in Sect. 9.5). However, as the amount of fine-tuning data gradually increases to 1%, 5%, and 10%, the performance gap between the two approaches narrows. It is worth noting that when the model is pretrained on ActiveMQ and fine-tuned with 5% or 10% of the data on other projects, SFFL exhibits better refactoring performance than SCG.

Therefore, we believe that SCG and SFFL have their own advantages and could complement each other in different situations. Furthermore, a relatively straightforward way to combine the strengths of both is to use SCG for *feature envy* detection

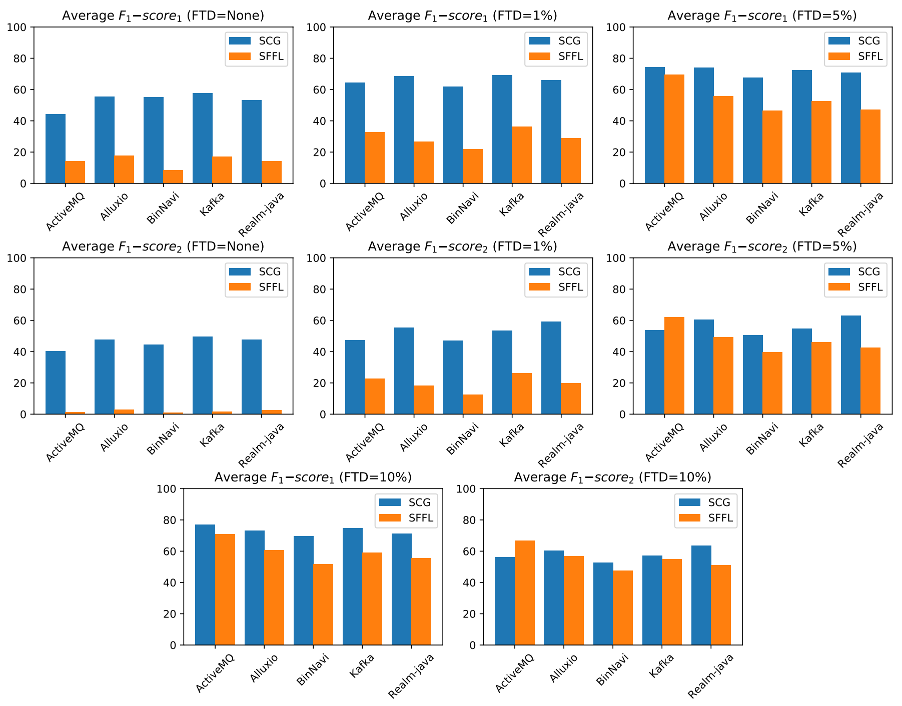


Fig. 15 Performance comparison in cross-project scenarios. FTD refers to Fine-tuning data. The x-axis represents the project on which the model is pre-trained, and the y-axis represents the average performance of the pre-trained model when detecting and refactoring in the other four projects

and SFFL for *feature envy* refactoring when the target project has 10% of fine-tuning data.

10 Conclusion and future work

In this paper, we present **SCG** (**S**MOTE **C**all **G**raph), a Graph Neural Network (GNN)-based approach, to address the *feature envy* detection problem. SCG formulates the detection task as a node binary classification problem on the method call graph. To address the issue of imbalanced positive and negative nodes in the graph, SCG utilizes GraphSMOTE to enhance the method call graph. Meanwhile, it introduces the concept of *calling strength* to quantify the strength of method invocations in the call graph. Additionally, SCG converts the method call graph into a method-class call graph representation. Finally, SCG recommends the smelly method to the external class with the highest *calling strength*.

Furthermore, we propose a holistic approach named **SFFL** (**S**ymmetric **F**eature **F**usion **L**earning) to directly address the *feature envy* refactoring problem. SFFL collects invocation, ownership, position and semantic information of all methods and classes from a project, and then encodes the information into four directed heterogeneous graphs. Symmetric Feature Fusion Learning is introduced to obtain the representations of methods and classes, so as to employ link prediction on them. To the best of our knowledge, we are the first to transform *feature envy* refactoring into edge reconstruction of method and class ownership heterogeneous graph. Unlike current deep learning approaches, we bypass the detection phase and directly refactor all methods in the project.

Last but not least, we propose three novel evaluation metrics that take into account both the detection and refactoring performance. Evaluation results suggest that the proposed approaches significantly outperform the competitors in both *feature envy* detection and refactoring. With the performance of our approaches in Java-developed projects, we believe they can be extended to projects developed by other object-oriented programming languages.

In the future, we will explore the impact of attribute access on *feature envy* refactoring. Additionally, we will attempt to extract richer semantic information, such as the bodies of methods and classes. Finally, we will develop a lightweight *feature envy* refactoring tool based on the pre-trained model.

Supplementary Information The online version contains supplementary material available at https://doi.org/10.1007/s10515-024-00476-3.

Acknowledgements This work was supported by National Natural Science Foundation of China under Grants 62372145 and 61902096, and Key Research and Development Program of Zhejiang Province under Grants 2023C03200 and 2023C03179. The authors would also like to thank the anonymous reviewers for their insightful comments.

References

Bahdanau, D., Cho, K., Bengio, Y.: Neural machine translation by jointly learning to align and translate. In: Bengio Y, LeCun Y (eds) 3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings, arXiv:1409.0473 (2015)

Bui, N.D.Q., Yu, Y., Jiang, L.: Infercode: Self-supervised learning of code representations by predicting subtrees. In: 43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021. IEEE, pp 1186–1197, https://doi.org/10.1109/ICSE43902.2021.00109, (2021)

Chawla, N.V., Bowyer, K.W., Hall, L.O., et al..: SMOTE: synthetic minority over-sampling technique. J. Artif. Intell. Res. **16**, 321–357 (2002). https://doi.org/10.1613/jair.953

Cunningham, W.: The wycash portfolio management system. OOPS Messenger **4**(2), 29–30 (1993). https://doi.org/10.1145/157710.157715

da Silva, Maldonado E., Shihab, E., Tsantalis, N.: Using natural language processing to automatically detect self-admitted technical debt. IEEE Trans Softw Eng **43**(11), 1044–1062 (2017)

Devlin, J., Chang, M., Lee, K., et al..: BERT: pre-training of deep bidirectional transformers for language understanding. In: Burstein J, Doran C, Solorio T (eds) Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers). Association for Computational Linguistics, pp 4171–4186, https://doi.org/10.18653/v1/n19-1423 (2019)

Fowler, M.: Refactoring - Improving the Design of Existing Code. Addison Wesley object technology series, Addison-Wesley, http://martinfowler.com/books/refactoring.html (1999)

Guo, X., Shi, C., Jiang, H.: Deep semantic-based feature envy identification. In: Internetware '19: The 11th Asia-Pacific Symposium on Internetware, Fukuoka, Japan, October 28-29, 2019. ACM, pp 19:1–19:6, https://doi.org/10.1145/3361242.3361257 (2019)

Guo, Z., Liu, S., Liu, J., et al..: How far have we progressed in identifying self-admitted technical debts? a comprehensive empirical study. ACM Trans Softw Eng Methodol **30**(4), 1–56 (2021)

Hadj-Kacem, M., Bouassida, N.: A multi-label classification approach for detecting test smells over java projects. J King Saud Univ Comput Inf Sci **34**(10), 8692–8701 (2022)

Hamilton, W.L., Ying, Z., Leskovec, J.: Inductive representation learning on large graphs. In: Guyon I, von Luxburg U, Bengio S, et al. (eds) Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA, pp 1024–1034, https://proceedings.neurips.cc/paper/2017/hash/5dd9db5e033da9c6fb5ba83c7a7ebea9-Abstract.html (2017)

Hochreiter, S., Schmidhuber, J.: Long short-term memory. Neural Comput **9**(8), 1735–1780 (1997). https://doi.org/10.1162/neco.1997.9.8.1735

Kingma, D.P., Ba, J.: Adam: A method for stochastic optimization. In: Bengio Y, LeCun Y (eds) 3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings, arXiv:1412.6980 (2015)

Kipf, T.N., Welling, M.: Semi-supervised classification with graph convolutional networks. In: 5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings. OpenReview.net, https://openreview.net/forum?id=SJU4ayYgl (2017)

Le, Q.V., Mikolov, T.: Distributed representations of sentences and documents. In: Proceedings of the 31th International Conference on Machine Learning, ICML 2014, Beijing, China, 21-26 June 2014, JMLR Workshop and Conference Proceedings, vol 32. JMLR.org, pp 1188–1196, http://proceedings.mlr.press/v32/le14.html (2014)

LeCun, Y., Bottou, L., Bengio, Y., et al..: Gradient-based learning applied to document recognition. Proc IEEE **86**(11), 2278–2324 (1998). https://doi.org/10.1109/5.726791

Liu, H., Xu, Z., Zou, Y.: Deep learning based feature envy detection. In: Huchard M, Kästner C, Fraser G (eds) Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018. ACM, pp 385–396, https://doi.org/10.1145/3238147.3238166 (2018)

Lozano, A., Mens, K., Portugal, J.: Analyzing code evolution to uncover relations. In: 2015 IEEE 2nd International Workshop on Patterns Promotion and Anti-patterns Prevention (PPAP), pp 1–4, https://doi.org/10.1109/PPAP.2015.7076847 (2015)

Ma, W., Yu, Y., Ruan, X., et al..: Pre-trained model based feature envy detection. In: 20th IEEE/ACM International Conference on Mining Software Repositories, MSR 2023, Melbourne, Australia, May 15-16, 2023. IEEE, pp 430–440, https://doi.org/10.1109/MSR59073.2023.00065 (2023)

Mikolov, T., Chen, K., Corrado, G., et al..: Efficient estimation of word representations in vector space. In: Bengio Y, LeCun Y (eds) 1st International Conference on Learning Representations, ICLR 2013, Scottsdale, Arizona, USA, May 2-4, 2013, Workshop Track Proceedings, arXiv:1301.3781 (2013)

Nandani, H., Saad, M., Sharma, T.: DACOS - A manually annotated dataset of code smells. In: 20th IEEE/ACM International Conference on Mining Software Repositories, MSR 2023, Melbourne, Australia, May 15-16, 2023. IEEE, pp 446–450,https://doi.org/10.1109/MSR59073.2023.00067 (2023)

Palomba, F., Bavota, G., Penta, M.D., et al..: On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. Empir Softw Eng **23**(3), 1188–1221 (2018). https://doi.org/10.1007/s10664-017-9535-z

Sales, V., Terra, R., Miranda, L.F., et al..: Recommending move method refactorings using dependency sets. In: Lämmel R, Oliveto R, Robbes R (eds) 20th Working Conference on Reverse Engineering, WCRE 2013, Koblenz, Germany, October 14-17, 2013. IEEE Computer Society, pp 232–241, https://doi.org/10.1109/WCRE.2013.6671298 (2013)

Schuster, M., Paliwal, K.K.: Bidirectional recurrent neural networks. IEEE Trans Signal Process **45**(11), 2673–2681 (1997). https://doi.org/10.1109/78.650093

Shahidi, M., Ashtiani, M., Nasrabadi, M.Z.: An automated extract method refactoring approach to correct the long method code smell. J Syst Softw **187**, 111221 (2022). https://doi.org/10.1016/j.jss.2022.111221

Sharma, T., Kessentini, M.: Qscored: A large dataset of code smells and quality metrics. In: 18th IEEE/ACM International Conference on Mining Software Repositories, MSR 2021, Madrid, Spain, May 17-19, 2021. IEEE, pp 590–594, https://doi.org/10.1109/MSR52588.2021.00080 (2021)

Sharma, T., Efstathiou, V., Louridas, P., et al..: Code smell detection by deep direct-learning and transfer-learning. J Syst Softw **176**, 110936 (2021). https://doi.org/10.1016/j.jss.2021.110936

Simon, F., Steinbrückner, F., Lewerentz, C.: Metrics based refactoring. In: Sousa P, Ebert J (eds) Fifth Conference on Software Maintenance and Reengineering, CSMR 2001, Lisbon, Portugal, March 14-16, 2001. IEEE Computer Society, pp 30–38, https://doi.org/10.1109/.2001.914965 (2001)

Terra, R., Valente, M.T., Miranda, S., et al..: Jmove: a novel heuristic and tool to detect move method refactoring opportunities. J Syst Softw **138**, 19–36 (2018). https://doi.org/10.1016/j.jss.2017.11.073

Tsantalis, N., Chatzigeorgiou, A.: Identification of move method refactoring opportunities. IEEE Trans Softw Eng **35**(3), 347–367 (2009). https://doi.org/10.1109/TSE.2009.1

Tsantalis, N., Chaikalis, T., Chatzigeorgiou, A.: Ten years of jdeodorant: Lessons learned from the hunt for smells. In: Oliveto R, Penta MD, Shepherd DC (eds) 25th International Conference on Software Analysis, Evolution and Reengineering, SANER 2018, Campobasso, Italy, March 20-23, 2018. IEEE Computer Society, pp 4–14, https://doi.org/10.1109/SANER.2018.8330192 (2018)

Vaswani, A., Shazeer, N., Parmar, N., et al..: Attention is all you need. In: Guyon I, von Luxburg U, Bengio S, et al. (eds) Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA, pp 5998–6008, https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html (2017)

Velickovic, P., Cucurull, G., Casanova, A., et al..: Graph attention networks. In: 6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings. OpenReview.net, https://openreview.net/forum?id=rJXMpikCZ (2018)

Wang, H., Liu, J., Kang, J., et al..: Feature envy detection based on bi-lstm with self-attention mechanism. In: Hu J, Min G, Georgalas N, et al. (eds) IEEE International Conference on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking, ISPA/BDCloud/SocialCom/SustainCom 2020, Exeter, United Kingdom, December 17-19, 2020. IEEE, pp 448–457, https://doi.org/10.1109/ISPA-BDCloud-SocialCom-SustainCom51426.2020.00082 (2020)

Yin, X., Shi, C., Zhao, S.: Local and global feature based explainable feature envy detection. In: IEEE 45th Annual Computers, Software, and Applications Conference, COMPSAC 2021, Madrid, Spain, July 12-16, 2021. IEEE, pp 942–951, https://doi.org/10.1109/COMPSAC51774.2021.00127 (2021)

Yu, D., Xu, Y., Weng, L., et al..: Detecting and refactoring feature envy based on graph neural network. In: IEEE 33rd International Symposium on Software Reliability Engineering, ISSRE 2022, Charlotte, NC, USA, October 31 - Nov. 3, 2022. IEEE, pp 458–469, https://doi.org/10.1109/ISSRE55969.2022.00051 (2022)

Zhang, M., Jia, J.: Feature envy detection with deep learning and snapshot ensemble. In: 9th International Conference on Dependable Systems and Their Applications, DSA 2022, Wulumuqi, China, August 4-5, 2022. IEEE, pp 215–223, https://doi.org/10.1109/DSA56465.2022.00037 (2022)

Zhao, S., Shi, C., Ren, S., et al..: Correlation feature mining model based on dual attention for feature envy detection. In: Peng R, Pantoja CE, Kamthan P (eds) The 34th International Conference on Software Engineering and Knowledge Engineering, SEKE 2022, KSIR Virtual Conference Center, USA, July 1 - July 10, 2022. KSI Research Inc., pp 634–639, https://doi.org/10.18293/SEKE2022-009 (2022)

Zhao, T., Zhang, X., Wang, S.: Graphsmote: Imbalanced node classification on graphs with graph neural networks. In: Lewin-Eytan L, Carmel D, Yom-Tov E, et al. (eds) WSDM '21, The Fourteenth ACM International Conference on Web Search and Data Mining, Virtual Event, Israel, March 8-12, 2021. ACM, pp 833–841, https://doi.org/10.1145/3437963.3441720 (2021)

Authors and Affiliations

Dongjin Yu[1] · Yihang Xu[1] · Lehui Weng[1] · Jie Chen[1] · Xin Chen[1] · Quanxin Yang[1,2]

✉ Dongjin Yu
yudj@hdu.edu.cn

Yihang Xu
yihangxu@hdu.edu.cn

Lehui Weng
192050142@hdu.edu.cn

Jie Chen
cjie@hdu.edu.cn

Xin Chen
chenxin4391@hdu.edu.cn

Quanxin Yang
quanxinyg@gmail.com

1 College of Computer Science and Technology, Hangzhou Dianzi University, Baiyang Street, Hangzhou 310000, Zhejiang, China

2 School of Information Engineering, Xuchang University, Xuchang, China