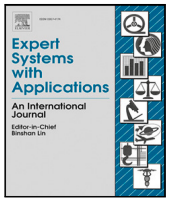
Expert Systems With Applications 258 (2024) 125164

Contents lists available at ScienceDirect

Expert Systems With Applications

journal homepage: www.elsevier.com/locate/eswa



Multilingual code refactoring detection based on deep learning

Tao Li[1], Yang Zhang [*,1]

School of Information Science and Engineering, Hebei University of Science and Technology, Shijiazhuang 050018, China

ARTICLE INFO

Keywords:
Refactoring detection
Deep learning
Code change
Multilingual code
Edit sequence

ABSTRACT

Refactoring is a critical process of improving the internal structure of the source code without altering its external behavior. Existing deep learning-based refactoring detection relies on commit messages to extract features. However, these commit messages are not trustful enough since some developers do not consistently record refactoring activities. Furthermore, current approaches are designed for a single programming language and lack multilingual refactoring support. To this end, this paper proposes *RefT5*, a multilingual code refactoring detection approach based on deep learning. Firstly, we select 110 real-world projects with Java and Python programming languages as a corpus to construct the dataset. Secondly, we extract features including commit messages, code changes, and refactoring types from these projects. *RefT5* generates edit sequences from code changes and takes refactoring types as labels. Thirdly, we employ *CodeT5* and *BiLSTM-attention* to extract semantic and structural features and generate feature vectors. Finally, the feature vectors are input into a classification layer to detect the refactoring type. The experimental results show that *RefT5* obtains 98.05% precision and 97.77% recall. Furthermore, compared with existing approaches, it improves precision by 51.61% and recall by 52.9% on average, demonstrating its effectiveness.

1. Introduction

Refactoring is a technique that improves the internal structure of a software system without changing its external behavior. It is widely applied in the software evolution process and becomes increasingly important in enhancing the readability and maintainability of the source code. Refactoring detection is crucial for developers because it can automatically detect and classify refactoring operations and improve code quality, maintainability, and performance. It helps developers better manage and maintain the source code, thus improving the reliability and efficiency of the software.

Researchers have developed several refactoring detection tools. Most refactoring detection tools (Dig, Manzoor, Johnson, & Nguyen, 2008; Prete, Rachatasumrit, Sudan, & Kim, 2010; Silva, da Silva, Santos, Terra, & Valente, 2020; Xing & Stroulia, 2005) rely on similarity thresholds. However, subjective factors can easily influence these thresholds, and it is difficult to adjust the thresholds according to differences between projects. To solve the issue of similarity threshold, RefactoringMiner (Tsantalis, Ketkar, & Dig, 2020) adopts a statement-matching algorithm based on an abstract syntax tree that does not require the user to specify the threshold value. However, RefactoringMiner only focuses on the Java programming language. In recent years, with the rapid development of deep learning, many researchers have begun to apply deep learning to refactoring. Some researchers (Krasniqi & Cleland-Huang, 2020; Tan & Bockisch, 2022) have combined deep learning with refactoring detection tools to improve accuracy. In addition, some works take commit messages as input and use deep learning techniques to predict refactoring types (Marmolejos, AlOmar, Mkaouer, Newman, & Ouni, 2021; Sagar, AlOmar, Mkaouer, Ouni, & Newman, 2021).

Although deep learning-based refactoring detection techniques have demonstrated significant effectiveness, they still face three major challenges. Firstly, existing detection tools (Dig et al., 2008; Prete et al., 2010; Tsantalis et al., 2020; Xing & Stroulia, 2005) only support Java programming language, with insufficient attention paid to Python and other programming languages. Since many modern software projects are developed using multiple languages, it is increasingly crucial to create a refactoring detection tool that supports a variety of programming languages. The multilingual support in these tools enhances their versatility and extends their applicability to a wider range of software project environments. Secondly, the choice of dataset directly impacts the precision of deep learning-based refactoring detection methods. Relying solely on commit messages from version control systems is insufficient and poses reliability issues (Murphy-Hill, Parnin, & Black, 2011). Finally, deep learning-based refactoring detection requires selecting appropriate models to handle complex code-related data. These models need to accurately obtain the syntactical structures of different

* Corresponding author.
E-mail addresses: litao@stu.hebust.edu.cn (T. Li), zhangyang@hebust.edu.cn (Y. Zhang).
[1] Co-first authors who contribute equally.

https://doi.org/10.1016/j.eswa.2024.125164
Received 9 February 2024; Received in revised form 14 July 2024; Accepted 20 August 2024
Available online 22 August 2024

programming languages and understand the rich semantic information embedded within the source code.

To solve the aforementioned challenges, we propose *RefT5*, a novel refactoring detection method based on deep learning. Firstly, we collect 110 projects written in Java and Python from Github and focus on five refactoring types including *Add Parameter*, *Change Return Type*, *Remove Parameter*, *Rename Method*, and *Rename Parameter*. We construct a dataset with 17,278 samples. Secondly, we extract commit messages from those projects, identify various refactoring activities, and collect code change data corresponding to each refactoring type. We construct edit sequences based on the code change and integrate these sequences with the extracted commit messages and other related data to form training samples. Finally, we utilize the pre-trained model CodeT5 (Wang, Wang, Joty and Hoi, 2021) and the BiLSTM-attention (Yu, Wang, Chen, & Chen, 2021) to extract deep semantic and structural features from the sample data and apply these features to the detection of refactoring types. Compared to existing technologies, *RefT5* has significantly improved the performance of refactoring detection after incorporating code change and edit sequences, increasing the precision and recall rates of refactoring detection by 51.61% and 52.9%, respectively.

The main contributions of this paper can be summarized as follows.

- A total of 17,278 samples are collected from 110 projects, covering two programming languages and five refactoring types including Add Parameter, Change Return Type, Remove Parameter, Rename Method, and Rename Parameter.
- We propose a refactoring detection method based on deep learning that utilizes pre-trained models and edit sequences to achieve multilingual code refactoring detection.
- We evaluate the effectiveness of *RefT5* by answering 5 research questions, demonstrating its effectiveness.

The remainder of the paper is structured as follows. Section 2 introduces the related work. Section 3 presents an overview of *RefT5*. Section 4 evaluates the effectiveness of *RefT5*. Conclusions are drawn in Section 5.

2. Related work

This section introduces refactoring detection tools, deep learning-based refactoring detection, and code change representation.

2.1. Refactoring detection tools

Refactoring detection tools enable developers to automatically detect and categorize refactoring types, improving code quality and maintainability.

UMLDiff (Xing & Stroulia, 2005) is an algorithm used for automatically detecting structural changes between different versions of object-oriented software designs. It takes two class models of Java software systems as input and performs reverse engineering on the corresponding code versions.

REF-FINDER (Prete et al., 2010) tool is an effective refactoring detection tool that can identify complex types of refactoring. The tool uses template logical rules to express each type of refactoring and utilizes a logic programming engine to convert template logical rules into logical queries in order to infer specific refactoring instances.

RefactoringCrawler (Dig et al., 2008) is a plugin integrated into the Eclipse development environment. It aims to automatically detect refactoring activities between different versions of Java components. It combines fast syntactic analysis with more expensive semantic analysis to identify refactorings. This improves the accuracy of refactoring detection in real-world software components. The tool leverages Shingles encoding techniques to quickly find similar fragments in source files. It also determines potential refactoring relationships between candidate entities by analyzing reference graphs.

RefDiff (Silva et al., 2020) is a multi-language refactoring detection tool that automatically identifies refactoring operations in code changes across different programming languages, filling a gap that has previously been mostly limited to Java. The tool's design is based on two key decisions: first, its refactoring detection algorithm relies on information from abstract syntax trees (CSTs), a data structure that represents the source code but omits language-specific details, thus enabling loose coupling between languages. Second, code similarity is computed at the source code level after disambiguation through information retrieval techniques, which further enhances language generalization.

RefactoringMiner (Tsantalis et al., 2020) is an advanced code refactoring detection tool that innovates by not relying on code similarity thresholds and focuses on identifying refactoring operations that occur between software commits, including low-level, method-internal refactorings such as variable renaming and extraction. The tool distinguishes between pure refactoring operations and non-refactoring operations by analyzing the additions, deletions, and changes of code commits, handling refactoring-induced code changes using unique abstraction and parameterization techniques, and combining syntax-aware Abstract Syntax Tree (AST) node substitution to effectively deal with overlapping refactorings and changes induced by code maintenance activities.

The aforementioned five refactoring detection tools are not only widely used in practical applications, but also the five most commonly mentioned tools in academic literature. Most refactoring detection tools (Dig et al., 2008; Prete et al., 2010; Xing & Stroulia, 2005) utilize similarity thresholds when detecting refactoring types. RefDiff (Silva et al., 2020) reduces its reliance on user-defined or calibrated similarity thresholds. However, the recall rate of certain types of refactoring decreases due to low threshold settings, indicating that adjusting the threshold in different scenarios to optimize results remains a factor to consider. Setting these thresholds requires empirical data or manual intervention and is easily influenced by subjective factors. Moreover, it is impractical to find a universal threshold for different projects. Different thresholds can significantly impact precision and recall (Dig & Johnson, 2006) rates, and software systems with different architectural styles and frameworks require appropriate thresholds based on specific circumstances (Aniche, Treude, Zaidman, Van Deursen, & Gerosa, 2016). In contrast, RefactoringMiner is a code refactoring detection tool based on the AST statement matching algorithm that does not require users to specify thresholds. It decomposes diff code into abstract syntax tree nodes and uses matching algorithms to match the parsed nodes with refactoring candidate nodes. Finally, it determines each candidate object's refactoring type based on its features. Although RefactoringMiner supports numerous widely applicable refactoring types, it only supports Java.

2.2. Deep learning-based refactoring detection

Murphy-Hill et al. (2011) pointed out that commit messages in version history are unreliable indicators of refactoring activities because developers do not consistently record refactoring activities in commit messages. Sagar et al. (2021) employed different supervising machine learning models and LSTM models to predict refactoring types in projects. The first experiment used commit messages as input, while the second experiment combined commit messages with code metrics as input. This demonstrated the effectiveness of code metrics in predicting refactoring types. It was found that using commit messages with limited vocabulary alone cannot meet the training requirements of machine learning models. Marmolejos et al. (2021) proposed a framework to automatically identify and categorize refactoring documentation patterns found in commit messages, referred to as self-affirmed refactorings (SAR) (AlOmar, Mkaouer, & Ouni, 2019). By transforming the detection of refactoring documents into a binary classification problem, the study explored effective combinations of different feature extraction

techniques with five distinct binary classification machine learning algorithms to achieve optimal performance.

Krasniqi and Cleland-Huang (2020) proposed a refactoring detection tool called CMMiner, which detects and classifies refactoring descriptions in commit messages. The experiment compared multiple machine learning classification algorithms and demonstrated that combining CMMiner and RefactoringMiner enhances the integrity of refactoring detection and provides refactoring justification. Tan and Bockisch (2022) proposed a method that encodes differences between nodes in a syntax tree as image data for neural network matching, with a feature matching network that improves the precision and recall of RefDiff 2.0 (Silva et al., 2020) while demonstrating good robustness.

Although the aforementioned methods have made progress in improving the accuracy and recall of refactoring detection, there are still some limitations. Due to the lack of consistency among developers in documenting refactoring activities, commit messages often lack information about refactoring, which limits the effectiveness of detection methods based on commit messages. In addition, the performance of methods that only target commit messages containing specific keywords is also affected when dealing with commit messages that do not contain these keywords.

2.3. Code change representation

Code changes involve additions, deletions, and modifications to the source code. RefactoringMiner utilizes code change information from diff as input to the matching algorithm. Moreover, code change information has been successfully adopted for various software engineering tasks, such as just-in-time comment updating (Huang, Yu, Fan, Zhou, & Li, 2022; Mi, Zhang, Tang, Ju, & Lan, 2023), just-in-time defect prediction (Pornprasit & Tantithamthavorn, 2021; Zeng, Zhang, Zhang, & Zhang, 2021; Zheng, Shen, & Chen, 2021), conflict resolution (Dinella et al., 2022; Svyatkovskiy et al., 2022), and code commit message generation (Jung, 2021; Nie et al., 2021; Wang, Xia et al., 2021). Code changes have two primary forms: serialized and structured representations. Serialized representation encodes different symbols or strings as numbers using techniques like word or character encoding. Structured representation converts symbols and strings into structured data, including abstract syntax trees (ASTs) (Cabrera Lozoya, Baumann, Sabetta, & Bezzi, 2021; Liu, Gao, Chen, Nie, & Liu, 2020; Yao, Xu, Yin, Sun, & Neubig, 2021) and code change graphs (Dong et al., 2022; Panthaplackel, Allamanis and Brockschmidt, 2021; Panthaplackel, Li, Gligoric and Mooney, 2021; Yin, Neubig, Allamanis, Brockschmidt, & Gaunt, 2018).

With the development of deep learning, researchers have employed various deep learning models to learn features from diff information and applied them to various software engineering tasks. Xu et al. (2019) proposed the CODISUM method for automatically generating source code commit messages to aid developers in better understanding program and software evolution. This approach employs a GRU model that takes diff as input, combines code structure and semantic information, and utilizes copy mechanisms to address out-of-vocabulary issues. To learn good representations for code modifications and comments under limited resources, Siow, Gao, Fan, Chen, and Liu (2020) designed a multi-level embedding (i.e., word embedding and character embedding) method (CORE) to represent the semantics provided by code changes and comments and then trained the information in the embedding layer through an attention-based deep learning model. Hoang, Dam, Kamei, Lo, and Ubayashi (2019) proposed an end-to-end deep learning framework called DeepJIT, which can automatically extract features from commit messages and code changes and use them to identify defects. Ciborowska and Damevski (2022) proposed a method for automatically retrieving change sets causing newly reported errors using popular BERT models to more accurately match the semantics in error report text with the induced change sets. Research has also been done to learn code change representations using pre-trained models (Lin, Liu, Zeng, Jiang, & Cleland-Huang, 2021; Nie et al., 2021; Svyatkovskiy et al., 2022; Zhou et al., 2021), which have achieved good results.

It is evident that the integration of deep learning technology with code change information has become a research hotspot in the field of software engineering. By utilizing deep learning to extract and analyze code change information, key features can be obtained, providing significant support for code maintenance, management, and testing. Inspired by the combination of code change information and deep learning techniques, our research aims to solve the unreliability issue of commit messages by extracting features related to refactoring types from code change data.

3. Refactoring detection method

The overview of the *RefT5* is presented in Fig. 1. Initially, we extract source code from 110 real-world Java and Python applications and gather information such as commit messages and refactoring types. Subsequently, data preprocessing is carried out for instances of refactoring detection. Then, we generate edit sequences based on the code change information and combine these sequences with commit messages and other relevant data to form data samples. These samples are used as training inputs for the deep learning model.

Code change information is fed into the CodeT5 (Wang, Wang et al., 2021) model, while commit messages and edit sequences are separately fed into the BiLSTM-attention (Yu et al., 2021) model. Refactoring types serve as the labels for the samples. The model is expected to output the refactoring type labels for the samples. After multiple iterations of training on the training dataset, a trained classifier is obtained. We have evaluated the performance of the classifier on the test set and provided the results of refactoring detection.

3.1. Data collection

To enable multilingual code refactoring detection in deep learning models, it is imperative to construct datasets from diverse programming languages. We need to select and clone open-source projects on GitHub. The Java projects are selected randomly from a dataset made available by Aniche, Maziero, Durelli, and Durelli (2020), while the Python projects come from Dilhara, Ketkar, Sannidhi, and Dig (2022). We randomly select projects and prioritize cloning those that contain a large number of commit messages. We ultimately chose 60 open-source Java projects and 50 open-source Python projects hosted on GitHub for our dataset.

During the construction of the Java dataset, the extraction of data from Java projects is facilitated by a refactoring detection tool RefactoringMiner. RefactoringMiner utilizes an AST (Abstract Syntax Tree)-based statement-matching algorithm for the identification of refactoring types. The data includes the commit_ID, refactoring types, and start line numbers of code blocks before and after refactoring. To accurately locate the modified code blocks within a file that may contain multiple changes, we initially identify the starting line numbers of code segments associated with the specified refactoring types. Subsequently, we employ PyDriller (Spadini, Aniche, & Bacchelli, 2018), a Python framework designed for parsing Git repositories, to retrieve the commit messages linked to each commit_ID as well as the code blocks that correspond to the identified starting line numbers.

The process of collecting the Python dataset is similar to that of the Java dataset. We select the refactoring detection tool PyRef (Atwi et al., 2021). This refactoring detection tool only recognizes five refactoring types, such as the Add Parameter and Rename Method. Therefore, the refactoring types in the Java dataset are also filtered to select the corresponding five types.

To enhance the precision of the model, data preprocessing is essential. The collected commit messages contain irrelevant information,

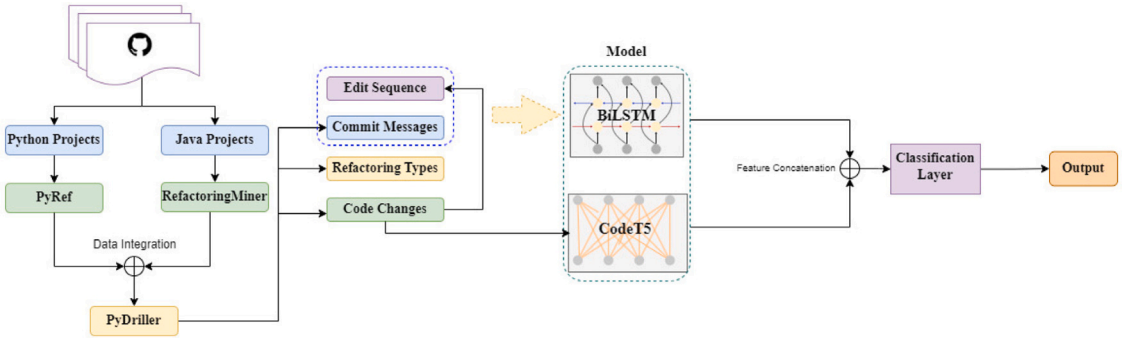


Fig. 1. Overview of *RefT5*.

such as comments and special characters. Moreover, the code before and after changes encompass single or multiple lines of comments. To mitigate noise in the data, we employ regular expressions to filter out these comments, special characters, and other redundant information.

The dataset consists of commit messages, edit sequences, diffs of code before and after refactoring, and refactoring type labels used for classification. These data serve as inputs for the model. In total, we have collected 17,278 samples.

To ensure the quality of our dataset, we design a two-step verification process. Initially, we verify the refactoring types by employing the change_type attribute from the PyDriller framework. The change_type parameter specifies the nature of the change, which includes categories such as *Added*, *Deleted*, *Modified*, and *Renamed*. Subsequently, for records that exhibit discrepancies with the anticipated refactoring types, we perform a manual inspection to confirm the data. Our dataset is publicly available at https://github.com/huatouguia98/dataset.

3.2. Edit sequence

The edit sequence (Pravilov, Bogomolov, Golubev, & Bryksin, 2021; Yin et al., 2018) is extensively used in fields such as computer science and bioinformatics to evaluate the similarity between two sequences. An edit sequence comprises a set of operations that transform one string into another.

Similar to the strategy used by Pravilov et al. (2021), we refine the edit representation (Yin et al., 2018). This approach selectively retains altered markers within the edit sequence while discarding those unmodified during code transformations. This refinement yields more abstract edit representation vectors and reduces the distance between vectors that are structurally akin. The advantage of this modification is the enhanced generalizability of the edit representation, making it suitable for various downstream tasks that involve processing code changes.

The original and modified codes in diff are two strings treated as token sequences. A deterministic difference algorithm based on the Levenshtein distance (Levenshtein, 1966) is applied to the two token sequences to construct the edit sequence.

The calculation of edit distance is achieved through a meticulous comparison between the character sequences of the source and target strings, thereby identifying the minimum number of editing steps required to transform the source string into the target string. In this process, four fundamental operation codes: 'insert', 'delete', 'replace', and 'equal', are precisely determined. This computational procedure typically employs dynamic programming strategies to incrementally construct a solution matrix that facilitates the determination of the optimal sequence of edits.

The derivation of operation codes within the context of edit distance computation is predicated upon the analysis of a matrix populated via dynamic programming. This matrix, which we denote as D, is systematically filled with values that represent the minimum number of edit operations required to transform the prefix of the source string s of length i into the prefix of the target string t of length j. The initialization of the matrix sets the stage for the subsequent derivation of edit operations. Specifically, the first row and column of the matrix D are initialized to correspond to the number of insertions and deletions needed to transform between an empty string and the respective prefixes of s and t. This forms the base conditions for the recursive computation that follows.

As iterating through the matrix and filling in the values of $D[i][j]$, we consider the costs of three elementary edit operations: insertion, deletion, and replacement. The cost of insertion is derived from the value $D[i][j-1]$ by adding one, signifying the addition of a character to transform $s[1...i]$ into $t[1...(j-1)]$. Conversely, the cost of deletion is obtained from the value $D[i-1][j]$ plus one, indicating the removal of a character to match $s[1...(i-1)]$ to $t[1...j]$. For replacement, if the characters $s[i]$ and $t[j]$ are not equal, the cost is the value at $D[i-1][j-1]$ plus one; otherwise, if the characters are equal, no additional cost is incurred, and the value remains $D[i-1][j-1]$. These operations are evaluated to determine the minimum cost at each cell of the matrix, encapsulating the essence of the edit distance algorithm.

The extraction of the operation codes, representing the specific edits required to transform string s into string t, is accomplished through a traceback process starting from $D[m][n]$, where m and n are the lengths of s and t, respectively. The traceback interprets the matrix's values to infer the preceding operations: a diagonal move from $D[i][j]$ to $D[i-1][j-1]$ indicates an 'equal' operation if $s[i]$ equals $t[j]$, or a 'replace' operation if they differ. A vertical move from $D[i][j]$ to $D[i-1][j]$ reflects a 'delete' operation and a horizontal move to $D[i][j-1]$ signals an 'insert' operation. Through this reverse iteration, the sequence of operation codes is elucidated, providing a granular view of the transformation process and unveiling the minimal edit sequence that underpins the edit distance between the two strings.

An example of an edit sequence is shown in Fig. 2, which contains a code block and three sequences. The code in red indicates the code before modification and the code in green indicates the code after modification. Sequence ① represents the edit sequence, Sequence ② is the token sequence before modification, and Sequence ③ is the token sequence after modification. Padding symbols (Ø) keep the sequence length consistent when adding or modifying. The three sequences are the same length, with each column corresponding. When the token in the 2nd row is the same as the token in the 3rd row, the 1st row uses "=" to indicate no change operation. When tokens in the 2nd and 3rd rows are different, the 1st row uses "↔" to indicate a substitution operation. When there is a token in the 2nd row but not in the 3rd, the 1st row uses "−" to indicate a deletion operation. When there is no token in the 2nd row but there is one in the 3rd row, the 1st row uses "+" to indicate the addition operation.

The generated edit sequence can be utilized for several downstream applications that involve processing code changes, such as generating commit messages and classifying code alterations. We employ the edit sequence in our refactoring detection process to more effectively discern the structural and semantic intricacies of refactoring operations.

```
1  public static Document parseBodyFragment(String bodyHtml, String baseUri) {
2    ...
3    List <Node> nodeList = parseFragment(bodyHtml, body, baseUri);
4    Node[] nodes = nodeList.toArray(new Node[0]);
5    - for (int i = nodes.length - 1; i > nodeList.size(); i--) {
6    + for (int i = nodes.length - 1; i > 0; i--) {
7        nodes[i].remove();
8    }
9    ...
10 }
```

①	=	=	=	=	=	=	=	=	=	=	<->	=	=	=	=	=
②	for	(	int	i	=	nodes.length	-	1	;	i	>	nodeList.size()	;	i	--	)	{
③	for	(	int	i	=	nodes.length	-	1	;	i	>	0	;	i	--	)	{

Fig. 2. An example of edit sequence.

3.3. Feature extraction

To achieve multilingual code refactoring detection, it is necessary to extract valuable features from the collected data. Consider two input code snippets, C_{del} representing the deleted code and C_{add} representing the added code, which is subject to examination by a version control system. The tokenizer function $\mathcal{T}$ of the CodeT5 model processes these code snippets and converts them into a sequence of token IDs. This process is designed to capture the syntactic and semantic differences in the code modifications. The tokenization process is formalized as follows:

Let $C_{\text{del}} = \{c_1, c_2, \ldots, c_p\}$ and $C_{\text{add}} = \{c'_1, c'_2, \ldots, c'_q\}$ be two sequences of tokens that represent the deleted and added code changes respectively. The tokenizer $\mathcal{T}$ augments these sequences with special tokens that inform the transformer model of the beginning and transition points between the code changes. The output sequence O is computed by Eq. (1):

$$\begin{aligned} O &= \mathcal{T}\left(C_{\text{del}}, C_{\text{add}}\right) \\ &= \left[ID_{\text{CLS}}, ID_{c_1}, \ldots, ID_{c_p}, ID_{\text{SEP}}, ID_{c'_1}, \ldots, ID_{c'_q}, ID_{\text{SEP}}\right] \end{aligned} \tag{1}$$

where ID_{CLS} is the token ID used to denote the start of the sequence for classification tasks, and ID_{SEP} is the separator token ID that delineates the boundary between the deleted and added code sequences. Each ID_{c_i} and $ID_{c'_j}$ corresponds to the unique numerical ID assigned to the respective token by the tokenizer's vocabulary.

To conform to the fixed-length input requirement of the transformer model, the output sequence O may be padded with a padding token ID_{PAD} to a pre-defined length L, or truncated if the initial length exceeds L:

$$O' = \begin{cases} O \cup \{ID_{\text{PAD}}, ID_{\text{PAD}}, \ldots, ID_{\text{PAD}}\} & \text{if length}(O) < L \\ \text{truncate}(O, L) & \text{if length}(O) > L \\ O & \text{if length}(O) = L \end{cases} \tag{2}$$

The processed sequence O' is then provided to the CodeT5 model for the downstream task of refactoring detection.

To capture the intricate details of code changes and associated commit messages, we utilize a bidirectional Long Short-Term Memory (BiLSTM) network that separately processes the code edit sequences $\{e_1, e_2, \ldots, e_T\}$ and commit messages $\{m_1, m_2, \ldots, m_N\}$. The features extracted from each sequence are independently obtained before any subsequent fusion or analysis. Furthermore, the hidden state $h_t^{(x)}$ is obtained by combining the forward hidden state $\overrightarrow{\text{LSTM}}_x\left(x_t, \overrightarrow{h_{t+1}^{(x)}}\right)$ and the backward hidden state $\overleftarrow{\text{LSTM}}_x\left(x_t, \overleftarrow{h_{t+1}^{(x)}}\right)$, which is computed by Eq. (3):

$$h_t^{(x)} = \begin{cases} [\overrightarrow{\text{LSTM}}_e(e_t, \overrightarrow{h_{t-1}^{(e)}}); \overleftarrow{\text{LSTM}}_e(e_t, \overleftarrow{h_{t+1}^{(e)}})] \\ [\overrightarrow{\text{LSTM}}_m(m_t, \overrightarrow{h_{t-1}^{(m)}}); \overleftarrow{\text{LSTM}}_m(m_t, \overleftarrow{h_{t+1}^{(m)}})] \end{cases} \tag{3}$$

where x denotes the code edits sequence(e) or commit messages (m). The bidirectional LSTM processes each sequence type separately, with the forward and backward hidden states combined through concatenation for each time step.

The attention mechanism (Vaswani et al., 2017) is then applied to the feature representations of code edits and commit messages. The context vector c_i is the result of applying the attention mechanism, which is computed by Eq. (4):

$$c_i = \sum_{j=1}^{n} \alpha_{ij} h_j \tag{4}$$

$$\alpha_{ij} = \frac{\exp\left(e_{ij}\right)}{\sum_{k=1}^{n} \exp\left(e_{ik}\right)} \tag{5}$$

$$e_{ij} = \tanh\left(W_h h_j + b\right) \tag{6}$$

where n represents the total number of LSTM outputs and h_j denotes the output of LSTM. The attention weights α_{ij} are calculated by applying the softmax function to the alignment scores e_{ij}. This ensures that the attention weights are normalized and represent the relative importance of each LSTM output. The alignment scores are determined using the tanh activation function and the weight matrix W_h.

As shown in Fig. 3, the example demonstrates the process of feature extraction. Firstly, the code information is tokenized by breaking down excessively long words into subword forms and adding start and end symbols. This results in a token sequence. Next, each token is converted into its corresponding token ID and passed into the model. After training, the model generates feature vectors. Finally, these feature vectors are fed into the classification layer, which produces a predicted vector representing the probability distribution for each class.

3.4. Model

We present a hybrid classifier model that integrates CodeT5 (Wang, Wang et al., 2021) with a BiLSTM-attention mechanism for multilingual refactoring detection.

An architectural overview of our model is presented in Fig. 4. It includes two primary components: the CodeT5 pre-trained model and the BiLSTM-attention mechanism. In the initial phase, code change information is fed into the CodeT5 pre-trained model which encodes these changes into vector representations. CodeT5 leverages a multi-layer bidirectional transformer network and a self-attention mechanism to extract the semantic features embedded within the code alterations. In the subsequent phase, commit messages and edit sequences are processed as textual data inputs to the BiLSTM network. This network encodes the textual information and employs an attention mechanism to discern and weigh the significance of features pertinent to refactoring operations. This approach allows the model to concentrate on the important information.

After processing the input data, the model employs a fully connected layer to refine the output of the CodeT5 model. The textual and code feature representations are amalgamated via a merging layer. The final predictions are produced by a classification layer. The input dimension of the classification layer corresponds to the output dimension of the merging layer, and its output dimension is equal to the number of refactoring types that the model aims to identify. Structurally, the classification layer consists of the ReLU activation function, the fully connected layer, and the dropout layer. The dropout layer helps mitigate the possible overfitting.

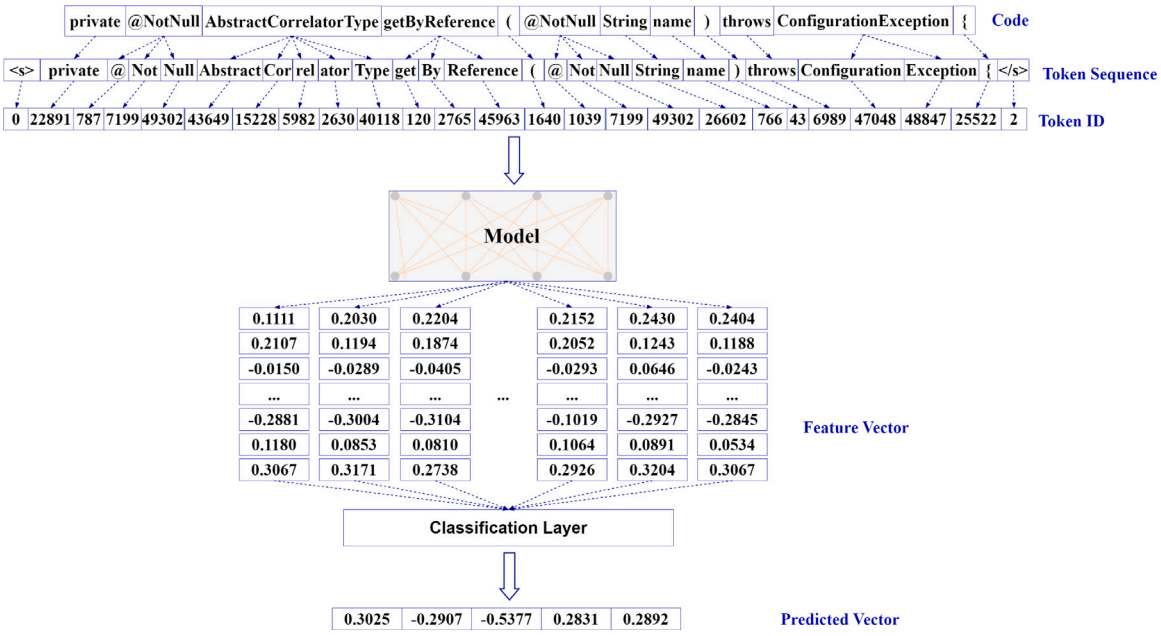


Fig. 3. An example of feature extraction.

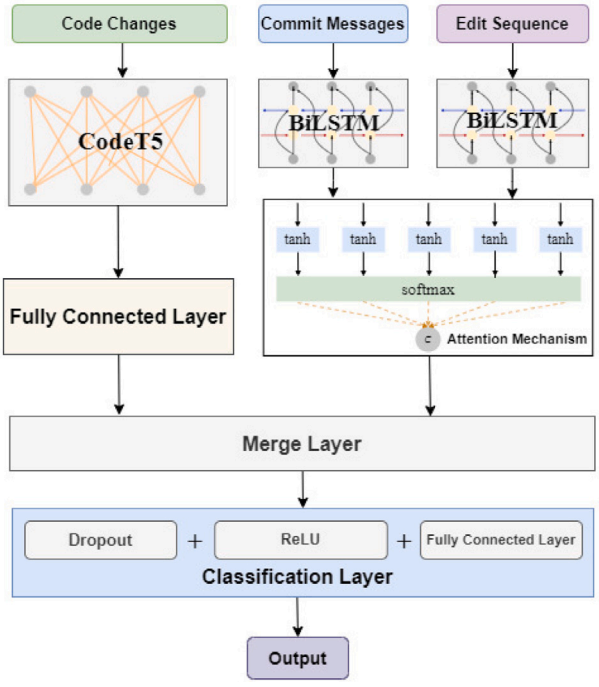


Fig. 4. The architecture of the *RefT5*'s model.

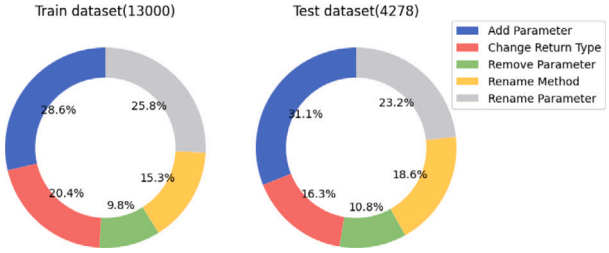


Fig. 5. Sample distribution.

4. Evaluation

This section introduces the dataset preparation, research questions, and evaluation metrics and analyzes the experimental results.

4.1. Dataset preparation

To assess the performance of the models, the datasets are systematically divided into training and test sets. Initially, the Java and Python datasets are delineated into their respective training and test sets. These individual sets are then consolidated: the Java and Python training sets are combined to form a hybrid training set for fine-tuning the models, while the test sets from both languages are unified to constitute a hybrid test set. The dataset structure comprises a hybrid dataset which includes the hybrid training and testing sets and dedicated test sets for Java and Python. Fig. 5 illustrates the distribution of each refactoring type across these datasets. The hybrid training set contains 13,000 samples, the hybrid test set includes 4278 samples, the Java test set holds 2352 samples, and the Python test set accounts for 1926 samples.

4.2. Research questions

We evaluate the effectiveness of *RefT5* by answering the following research questions (RQs):

RQ1 How effective is *RefT5* in multilingual code refactoring detection?

RQ2 How does the edit sequence improve the model's performance?

RQ3 How effective is *RefT5* compared to the existing work?

RQ4 How effective is *RefT5* compared to machine learning models?

RQ5 How effective is *RefT5* compared to other pre-trained models?

4.3. Evaluation metrics

For individual assessment of each refactoring type, we utilize precision, recall, and the F1 score as our evaluation metrics. To gauge the

overarching performance of *RefT5* on datasets with class imbalances, we resort to weighted average metrics, which factor in the varying class distributions.

Precision, recall, and the F1 score serve as fundamental metrics for assessing classification efficacy. These metrics are derived according to the formulas presented in Eqs. (7)–(9):

$$Precision = \frac{TP}{TP + FP} \tag{7}$$

$$Recall = \frac{TP}{TP + FN} \tag{8}$$

$$F1 = \frac{2 * Precision * Recall}{Precision + Recall} \tag{9}$$

The weighted average represents the weighted mean of the Precision, Recall, and F1 scores for each class, with the weights proportional to the support (the number of true instances) for each class in the dataset. The computation of these evaluation metrics adheres to the formulas delineated in Eqs. (10)–(12).

$$Precision_w = \frac{\sum_{i=1}^{n} support_i * precision_i}{\sum_{i=1}^{n} support_i} \tag{10}$$

$$Recall_w = \frac{\sum_{i=1}^{n} support_i * recall_i}{\sum_{i=1}^{n} support_i} \tag{11}$$

$$F1_w = \frac{\sum_{i=1}^{n} support_i * F1_i}{\sum_{i=1}^{n} support_i} \tag{12}$$

where $support_i$ is the number of samples in the dataset for the ith class, $precision_i$ is the precision for the ith class, $recall_i$ is the recall for the ith class, and $F1_i$ is the $F1$ score for the ith class.

4.4. Results

4.4.1. Results for RQ1

To answer RQ1, we evaluated the *RefT5* on the Java, Python, and hybrid test sets.

The evaluation results for *RefT5* are presented in Table 1. For the Java test set, *RefT5* demonstrates superior proficiency, attaining a weighted precision of 97.67%, a weighted recall of 97.66%, and a weighted F1 score of 97.66% over a set of 2352 test instances, demonstrating good performance of *RefT5*.

The performance of *RefT5* on the Python test set surpassed its Java results, obtaining a weighted precision of 98.40%, a weighted recall of 98.39%, and a weighted F1 score of 98.39% across a total of 1926 samples. These results affirm the enhanced efficacy of *RefT5* and its adaptability to the Python programming language.

For the hybrid test set comprising 4278 samples with Java and Python languages, *RefT5* demonstrates robust generalization capabilities. The model achieves a weighted precision of 98.07%, a weighted recall of 98.01%, and a weighted F1 score of 98.04%. These metrics attest to *RefT5*'s consistent high performance and its adeptness at code comprehension and analysis within a multilingual context.

In a variety of test datasets, *RefT5* has demonstrated superior performance. Its effectiveness is particularly evident in the excellent scores it achieves on weighted evaluation metrics. The underlying reason for this success is *RefT5*'s ingenious integration of the CodeT5 model with a Bi-LSTM model that takes a sequence of code edits as input, which also incorporates an Attention mechanism. CodeT5 is built on the T5 framework, specifically designed for understanding and generating tasks in programming languages, and is trained on a large codebase through self-supervised learning methods, aiming to master the syntax and semantics of code. Particularly when dealing with code change information in multiple programming languages, CodeT5 can rely on its exceptional context encoding capability to precisely identify subtle differences in code. This ability stems from its in-depth pretraining on the structures of various programming languages.

Table 1
The performance of the model on different programming languages.

Test set	Number	$Precision_w$	$Recall_w$	$F1_w$
Java	2352	97.67%	97.66%	97.66%
Python	1926	98.40%	98.39%	98.39%
Hybrid	4278	98.07%	98.01%	98.04%

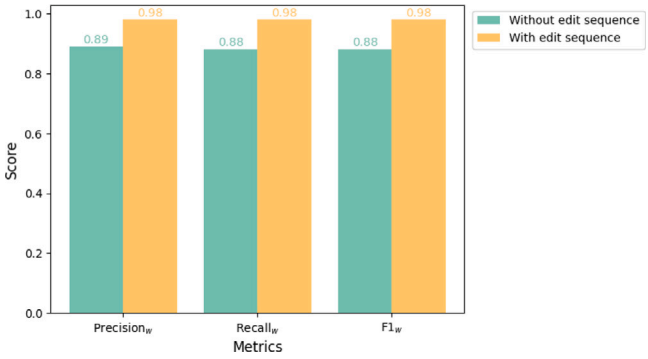


Fig. 6. The effect of edit sequence on performance.

On the other hand, the Bi-LSTM model effectively captures the contextual relationships of code changes through its bidirectional architecture, while the integrated Attention mechanism enables the model to focus on key code changes. The combination of these two technologies significantly improves the model's accuracy in identifying refactoring activities in the history of code edits.

By combining the multilingual analysis capabilities of CodeT5 with the sequence processing strengths of Bi-LSTM and Attention mechanisms, *RefT5* can deeply analyze code changes on multiple dimensions, significantly enhancing performance in refactoring detection tasks in multilingual programming environments. This integrated strategy not only improves the accuracy of refactoring detection but also enhances the efficiency of the overall analysis.

4.4.2. Results for RQ2

To answer RQ2, we built two datasets containing edit sequences and those without edit sequences.

Fig. 6 depicts a comparative evaluation of the *RefT5* model on two distinct datasets. When utilizing a dataset that includes edit sequences, the weighted precision, recall, and F1 score of *RefT5* exhibit increases of 9% (=98%–89%), 10% (=98%–88%), and 10% (=98%–88%), respectively, in comparison to the dataset devoid of such sequences. The primary reason for this enhancement is that the integration of special characters to denote edit operations enables the model to more accurately capture the nuances of code modifications. The employment of these special characters allows RefT5 to differentiate between various edit types and to develop more informative and representative embeddings.

From the experimental results, the model's performance has improved significantly after using edit sequences. This shows that edit sequences can effectively construct distributed representations of code changes and help the model learn better code features, thereby improving the model's precision.

4.4.3. Results for RQ3

To answer RQ3, we compared *RefT5* with the refactoring detection method proposed by Sagar et al. (2021). We chose the work of Sagar et al. to compare with our approach because their work uses commit messages as inputs to a deep learning model to predict refactoring activities, and their experimental results exemplify the unreliability of commit messages, which was one of the motivations for our research. The method introduced by Sagar et al. takes commit messages as input

Table 2
The comparison between RefT5 and Sagar et al.'s work (Sagar et al., 2021).

Refactoing type	Sagar et al.					*RefT5*				
	TP	FP	FN	Precision (%)	Recall (%)	TP	FP	FN	Precision (%)	Recall (%)
Add parameter	779	760	550	50.61%	58.62%	1299	43	30	96.80%	97.74%
Change return type	266	370	430	41.82%	38.22%	690	1	6	99.86%	99.14%
Remove parameter	158	191	305	45.27%	34.12%	445	10	18	97.80%	96.11%
Rename method	439	369	357	54.33%	55.15%	790	8	6	99.00%	99.25%
Rename parameter	380	566	614	40.17%	38.23%	960	32	34	96.77%	96.60%
Average	–	–	–	46.44%	44.87%	–	–	–	98.05%	97.77%

and utilizes an LSTM model. Table 2 presents a performance comparison of the two methods in detecting five refactoring types, which include adding parameters, changing return types, deleting parameters, renaming methods, and renaming parameters.

The data presented in the table clearly illustrates the marked advantage of *RefT5* in detecting refactoring activities. For example, regarding the "Add Parameter" refactoring type, *RefT5* attained 96.80% precision and 97.74% recall. This performance is substantially higher than the 50.61% precision and 58.62% recall achieved by the method developed by Sagar et al. This trend of *RefT5*'s superiority is consistent across various types of refactoring, with the "Change Return Type" category, in particular, showcasing near-perfect 99.86% precision and 99.14% recall, respectively.

The experimental results reveal that *RefT5* significantly outperforms the approach presented by Sagar et al. (2021), showing considerable improvements in precision and recall by 51.61% and 52.9%, respectively. This superior performance can likely be attributed to the limitations of Sagar et al.'s methodology, which relies solely on commit messages as input. This reliance may result in the derivation of features that are either insufficient or not sufficiently dependable. In contrast, *RefT5* leverages the wealth of information present in code modifications, enabling the extraction of a more substantial and reliable feature set. This approach substantially enhances the accuracy of refactoring detection.

4.4.4. Results for RQ4

To answer RQ4, we compare the *RefT5* model with six machine learning models: Support Vector Machine (SVM) (Pisner & Schnyer, 2020), Random Forest (RF) (Schonlau & Zou, 2020), Naive Bayes (NB) (Chen, Webb, Liu, & Ma, 2020), Decision Tree (DT) (Charbuty & Abdulazeez, 2021), Logistic Regression (LR) (Zou, Hu, Tian, & Shen, 2019), and Extreme Gradient Boosting Tree (XGBoost) (Wang, Deng, & Wang, 2020). All models are trained and tested using the hybrid dataset. Table 3 shows the results of the comparison with the machine learning models.

The findings reveal that the foundational machine learning models—Logistic Regression (LR), Decision Trees (DT), and Support Vector Machines (SVM)—demonstrate a fair level of performance, with each scoring approximately 70% across the precision, recall, and F1 metrics. In contrast, the Naive Bayes (NB) classifier, despite achieving 71% precision, shows a marked deficiency in recall and F1 score, which are only 44% and 45% respectively. This discrepancy highlights a pronounced imbalance in the NB classifier's capability to accurately identify true positives in conjunction with maintaining a minimal rate of false positives.

Advanced ensemble algorithms, including XGBoost and Random Forest (RF), have exhibited enhanced performance metrics. Specifically, XGBoost surpasses all other traditional models, securing precision, recall, and F1 scores of 79%, 78%, and 79%, respectively. Meanwhile, RF also delivers commendable outcomes, albeit marginally less impressive than XGBoost, with 76% precision, 73% recall, and 74% F1 score.

RefT5 demonstrates a notable edge over conventional machine learning methodologies, attaining remarkable scores across various metrics—weighted precision, recall, and F1—all reaching 98%. This marks a substantial enhancement, with the F1 score exhibiting a 19% increase in comparison to XGBoost, the most proficient among the conventional contenders. Such outstanding performance by *RefT5* is indicative of its robustness and its efficiency in identifying code refactoring patterns. This efficacy is likely due to its advanced architectural design, which is adept at discerning intricate patterns and dependencies within the dataset.

Table 3
The performance comparison with machine learning models.

Model	Precision_w	Recall_w	F1_w
LR	70%	69%	70%
DT	70%	70%	70%
NB	71%	44%	45%
XGBoost	79%	78%	79%
RF	76%	73%	74%
SVM	72%	71%	70%
RefT5	98%	98%	98%

The results underscore the capabilities of *RefT5* in identifying refactoring instances, clearly delineating its superiority over traditional machine learning models.

4.4.5. Results for RQ5

To answer RQ5, we conducted a comparative analysis of *RefT5* against other pre-trained models such as CodeBERT (Feng et al., 2020), GraphCodeBERT (Guo et al., 2020), UniXcoder (Guo et al., 2022), CodeReviewer (Li et al., 2022), and CodeT5+ (Wang et al., 2023) in the task of code refactoring detection. The results of this assessment are systematically delineated in Table 4.

Among the evaluated models, *RefT5* stands out with exemplary performance metrics, achieving a weighted precision of 98.07%, along with a weighted recall of 98.01%, and culminating in a weighted F1 score of 98.04%. These statistics not only underscore *RefT5*'s accuracy in identifying various types of refactoring but also its exceptional recall ability, which implies a minimal rate of overlooked refactoring instances.

The enhancements in the performance of *RefT5* are attributed to several key improvements in the T5 architecture made by the CodeT5 model, which is specifically tailored for better capturing the semantics of code. These improvements include a more complex understanding of the syntax and structure of programming languages, the integration of domain-specific knowledge that helps identify types of refactoring, and the optimization of the model's attention mechanism to more effectively discern the subtle nuances of the code context. These improvements collectively lead to an increase in precision and recall metrics.

In conclusion, the outcomes of our experimental analysis affirm the efficacy of *RefT5* and illuminate its prospective utility in automating the identification of code refactoring instances.

4.5. Discussion

We propose a cross-language refactoring detection method that effectively addresses the unreliability of relying on commit messages by integrating edit sequences with code change information. Particularly in Java and Python, two widely used programming languages, our

Table 4
The performance comparison with deep learning models.

Model	$Precision_w$	$Recall_w$	$F1_w$
CodeBERT	96.27%	96.42%	96.32%
GraphCodeBERT	97.01%	96.30%	96.61%
UniXcoder	93.54%	93.91%	93.71%
CodeReviewer	96.31%	95.95%	96.03%
CodeT5+	95.34%	94.46%	94.81%
RefT5	98.07%	98.01%	98.04%

method demonstrates significant results and provides a useful reference for cross-language refactoring detection.

Our approach is able to capture the patterns and intentions of code refactorings more accurately by introducing edit sequences. Edit sequences directly reflect the modification operations performed by developers on the code, and this directness improves the accuracy of refactoring identification compared to indirect inferences relying on commit messages. Further, combined with the ability to pre-trained models in CodeT5, we not only cross the language barrier to achieve refactoring detection in Java and Python but also leverage the contextual information in large-scale corpora to enhance the depth of understanding and generalization of the models.

In the daily practice of software development, where cross-language projects are increasing, our approach can effectively help teams identify refactoring behaviors, whether for code review or refactoring decision support. It can improve development efficiency and reduce errors caused by refactoring. In addition, for those legacy systems that need to be maintained for a long time, the method can help maintainers better understand the code evolution history and provide data support for refactoring decisions.

4.6. Interaction test

In our interaction test, we invited two professionals from industry and members of the experimental team to participate in the test. During the interaction test, the industry representatives pointed out that our method accurately recognizes the five refactoring types: Add Parameter, Change Return Type, Remove Parameter, Rename Method, and Rename Parameter. However, they also pointed out the limitation that our model is currently limited to these five refactoring types and lacks the ability to cover a wider range of refactoring types.

On the other hand, the experimental team conducted tests on multiple software projects. Although our method showed efficiency optimization when dealing with large-scale projects, the data preprocessing stage, which involves converting all data records into a model-compatible format, was time-consuming. Additionally, the step of constructing edit sequences was identified as a factor affecting the overall detection rate. Through the interaction test session, we deeply realized that, in order to enhance the practicality and competitiveness of this method, accelerating the data processing flow and expanding the detection range of refactoring types have become urgent key issues to be addressed. This indicates that future work should focus on further optimization of algorithm efficiency and the expansion of refactoring type detection capabilities.

4.7. Threats to validity

This section analyzes potential factors that could threaten the validity of experimental results.

Internal validity: The RefactoringMiner tool has demonstrated a high precision rate of 98% and a recall rate of 87%. In contrast, the PYREF refactoring detection tool has shown a precision rate of only 89.60% and a recall rate of 76.10%, with both tools exhibiting instances of false positives. Although we have implemented change_type parameter filtering within the PyDriller tool and manual review methods to reduce the proportion of false reports, it is still not possible to completely guarantee the elimination of all errors.

External validity: PYREF mainly detects refactoring at the method level and requires more work to extend support for class-level and field-level refactoring. Therefore, *RefT5* currently can only detect five common refactoring types, and its detection precision and recall may vary for other refactoring types, requiring further research. Additionally, our collected dataset only includes the Java and Python programming languages. Adapting RefT5 to other programming languages could affect the precision of refactoring detection, necessitating further study. To detect more refactoring types and support a broader range of programming languages, we need to refine *RefT5* and expand the dataset.

5. Conclusions

This paper proposes a novel approach called *RefT5* for multilingual code refactoring detection based on deep learning. It utilizes CodeT5 to delve into the semantic dimensions within code changes and integrates a BiLSTM-attention mechanism to extract features from edit sequences and commit messages. *RefT5* is able to identify five different types of refactoring operations. During constructing the dataset, we selected 110 projects based on Java and Python languages, aiming to achieve automated detection of multilingual refactoring. *RefT5* can process code in multiple programming languages and capture the characteristics of code changes between these languages through edit sequences. To evaluate the effectiveness of the *RefT5*, we conduct experiments to address five research questions. The experimental results demonstrate that *RefT5* obtains good performance in refactoring detection for both Java and Python programming languages. The future works include expanding our method to accommodate a broader spectrum of refactoring types as well as a wider range of programming languages, thereby further enhancing its efficiency. Furthermore, we will consider conducting detailed comparative analyses with large language models and exploring the application of our method to additional areas of software development to achieve more extensive technological dissemination and application.

CRediT authorship contribution statement

Tao Li: Original writing, Conducting experimentation. **Yang Zhang:** Proposing the idea, Experimental analysis, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

Declaration of Generative AI and AI-assisted technologies in the writing process

During the preparation of this work the author(s) used Grammarly in order to correct the English grammar. After using this tool/service, the authors reviewed and edited the content as needed and take full responsibility for the content of the publication.

Acknowledgments

The authors would like to thank the insightful comments and suggestions of those anonymous reviewers, which have improved the presentation. This work is partially supported by the Natural Science Foundation of Hebei under grant No. F2023208001 and the Overseas High-level Talent Foundation of Hebei under grant No. C20230358.

References

AlOmar, E., Mkaouer, M. W., & Ouni, A. (2019). Can refactoring be self-affirmed? an exploratory study on how developers document their refactoring activities in commit messages. In *2019 IEEE/ACM 3rd international workshop on refactoring* (pp. 51–58). IEEE.

Aniche, M., Maziero, E., Durelli, R., & Durelli, V. H. (2020). The effectiveness of supervised machine learning algorithms in predicting software refactoring. *IEEE Transactions on Software Engineering*, *48*(4), 1432–1450.

Aniche, M., Treude, C., Zaidman, A., Van Deursen, A., & Gerosa, M. A. (2016). SATT: Tailoring code metric thresholds for different software architectures. In *2016 IEEE 16th international working conference on source code analysis and manipulation* (pp. 41–50). IEEE.

Atwi, H., Lin, B., Tsantalis, N., Kashiwa, Y., Kamei, Y., Ubayashi, N., et al. (2021). PyRef: refactoring detection in python projects. In *2021 IEEE 21st international working conference on source code analysis and manipulation* (pp. 136–141). IEEE.

Cabrera Lozoya, R., Baumann, A., Sabetta, A., & Bezzi, M. (2021). Commit2vec: Learning distributed representations of code changes. *SN Computer Science*, *2*(3), 150.

Charbuty, B., & Abdulazeez, A. (2021). Classification based on decision tree algorithm for machine learning. *Journal of Applied Science and Technology Trends*, *2*(01), 20–28.

Chen, S., Webb, G. I., Liu, L., & Ma, X. (2020). A novel selective naïve Bayes algorithm. *Knowledge-Based Systems*, *192*, Article 105361.

Ciborowska, A., & Damevski, K. (2022). Fast changeset-based bug localization with BERT. In *Proceedings of the 44th international conference on software engineering* (pp. 946–957).

Dig, D., & Johnson, R. (2006). Automated upgrading of component-based applications. In *Companion to the 21st ACM SIGPLAN symposium on object-oriented programming systems, languages, and applications* (pp. 675–676).

Dig, D., Manzoor, K., Johnson, R. E., & Nguyen, T. N. (2008). Effective software merging in the presence of object-oriented refactorings. *IEEE Transactions on Software Engineering*, *34*(3), 321–335.

Dilhara, M., Ketkar, A., Sannidhi, N., & Dig, D. (2022). Discovering repetitive code changes in python ML systems. In *Proceedings of the 44th international conference on software engineering* (pp. 736–748).

Dinella, E., Mytkowicz, T., Svyatkovskiy, A., Bird, C., Naik, M., & Lahiri, S. (2022). Deepmerge: Learning to merge programs. *IEEE Transactions on Software Engineering*, *49*(4), 1599–1614.

Dong, J., Lou, Y., Zhu, Q., Sun, Z., Li, Z., Zhang, W., et al. (2022). FIRA: fine-grained graph-based code change representation for automated commit message generation. In *Proceedings of the 44th international conference on software engineering* (pp. 970–981).

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., et al. (2020). Codebert: A pre-trained model for programming and natural languages. arXiv preprint arXiv:2002.08155.

Guo, D., Lu, S., Duan, N., Wang, Y., Zhou, M., & Yin, J. (2022). Unixcoder: Unified cross-modal pre-training for code representation. arXiv preprint arXiv:2203.03850.

Guo, D., Ren, S., Lu, S., Feng, Z., Tang, D., Liu, S., et al. (2020). Graphcodebert: Pre-training code representations with data flow. arXiv preprint arXiv:2009.08366.

Hoang, T., Dam, H. K., Kamei, Y., Lo, D., & Ubayashi, N. (2019). DeepJIT: an end-to-end deep learning framework for just-in-time defect prediction. In *2019 IEEE/ACM 16th international conference on mining software repositories* (pp. 34–45). IEEE.

Huang, J., Yu, H., Fan, G., Zhou, Z., & Li, M. (2022). Improving just-in-time comment updating via AST edit sequence. *International Journal of Software Engineering and Knowledge Engineering*, *32*(10), 1455–1476.

Jung, T.-H. (2021). Commitbert: Commit message generation using pre-trained programming language model. arXiv preprint arXiv:2105.14242.

Krasniqi, R., & Cleland-Huang, J. (2020). Enhancing source code refactoring detection with explanations from commit messages. In *2020 IEEE 27th international conference on software analysis, evolution and reengineering* (pp. 512–516). IEEE.

Levenshtein, V. (1966). Binary codes capable of correcting deletions, insertions, and reversals. *Vol. 10*, In *Soviet physics doklady* (pp. 707–710). Soviet Union.

Li, Z., Lu, S., Guo, D., Duan, N., Jannu, S., Jenks, G., et al. (2022). CodeReviewer: Pre-training for automating code review activities. arXiv e-prints, arXiv–2203.

Lin, J., Liu, Y., Zeng, Q., Jiang, M., & Cleland-Huang, J. (2021). Traceability transformed: Generating more accurate links with pre-trained bert models. In *2021 IEEE/ACM 43rd international conference on software engineering* (pp. 324–335). IEEE.

Liu, S., Gao, C., Chen, S., Nie, L. Y., & Liu, Y. (2020). ATOM: Commit message generation based on abstract syntax tree and hybrid ranking. *IEEE Transactions on Software Engineering*, *48*(5), 1800–1817.

Marmolejos, L., AlOmar, E. A., Mkaouer, M. W., Newman, C., & Ouni, A. (2021). On the use of textual feature extraction techniques to support the automated detection of refactoring documentation. *Innovations in Systems and Software Engineering*, 1–17.

Mi, X., Zhang, J., Tang, Y., Ju, Y., & Lan, J. (2023). Boosting just-in-time code comment updating via programming context and refactor. *International Journal of Software Engineering and Knowledge Engineering*, 1–31.

Murphy-Hill, E., Parnin, C., & Black, A. P. (2011). How we refactor, and how we know it. *IEEE Transactions on Software Engineering*, *38*(1), 5–18.

Nie, L. Y., Gao, C., Zhong, Z., Lam, W., Liu, Y., & Xu, Z. (2021). Coregen: Contextualized code representation learning for commit message generation. *Neurocomputing*, *459*, 97–107.

Panthaplackel, S., Allamanis, M., & Brockschmidt, M. (2021). Copy that editing sequences by copying spans. *Vol. 35*, In *Proceedings of the AAAI conference on artificial intelligence* (pp. 13622–13630).

Panthaplackel, S., Li, J. J., Gligoric, M., & Mooney, R. J. (2021). Deep justintime inconsistency detection between comments and source code. *Vol. 35*, In *Proceedings of the AAAI conference on artificial intelligence* (pp. 427–435).

Pisner, D. A., & Schnyer, D. M. (2020). Support vector machine. In *Machine learning* (pp. 101–121). Elsevier.

Pornprasit, C., & Tantithamthavorn, C. K. (2021). JITLine: A simpler, better, faster, finer-grained just-in-time defect prediction. In *2021 IEEE/ACM 18th international conference on mining software repositories* (pp. 369–379). IEEE.

Pravilov, M., Bogomolov, E., Golubev, Y., & Bryksin, T. (2021). Unsupervised learning of general-purpose embeddings for code changes. In *Proceedings of the 5th international workshop on machine learning techniques for software quality evolution* (pp. 7–12).

Prete, K., Rachatasumrit, N., Sudan, N., & Kim, M. (2010). Template-based reconstruction of complex refactorings. In *2010 IEEE international conference on software maintenance* (pp. 1–10). IEEE.

Sagar, P. S., AlOmar, E. A., Mkaouer, M. W., Ouni, A., & Newman, C. D. (2021). Comparing commit messages and source code metrics for the prediction refactoring activities. *Algorithms*, *14*(10), 289.

Schonlau, M., & Zou, R. Y. (2020). The random forest algorithm for statistical learning. *The Stata Journal*, *20*(1), 3–29.

Silva, D., da Silva, J. P., Santos, G., Terra, R., & Valente, M. T. (2020). Refdiff 2.0: A multi-language refactoring detection tool. *IEEE Transactions on Software Engineering*, *47*(12), 2786–2802.

Siow, J. K., Gao, C., Fan, L., Chen, S., & Liu, Y. (2020). Core: Automating review recommendation for code changes. In *2020 IEEE 27th international conference on software analysis, evolution and reengineering* (pp. 284–295). IEEE.

Spadini, D., Aniche, M., & Bacchelli, A. (2018). Pydriller: Python framework for mining software repositories. In *Proceedings of the 2018 26th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering* (pp. 908–911).

Svyatkovskiy, A., Fakhoury, S., Ghorbani, N., Mytkowicz, T., Dinella, E., Bird, C., et al. (2022). Program merge conflict resolution via neural transformers. In *Proceedings of the 30th ACM joint European software engineering conference and symposium on the foundations of software engineering* (pp. 822–833).

Tan, L., & Bockisch, C. (2022). Diff feature matching network in refactoring detection. In *2022 29th Asia-Pacific software engineering conference* (pp. 199–208). IEEE.

Tsantalis, N., Ketkar, A., & Dig, D. (2020). RefactoringMiner 2.0. *IEEE Transactions on Software Engineering*, *48*(3), 930–950.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., et al. (2017). Attention is all you need. *Advances in Neural Information Processing Systems*, *30*.

Wang, C., Deng, C., & Wang, S. (2020). Imbalance-XGBoost: leveraging weighted and focal losses for binary label-imbalanced classification with XGBoost. *Pattern Recognition Letters*, *136*, 190–197.

Wang, Y., Le, H., Gotmare, A. D., Bui, N. D., Li, J., & Hoi, S. C. (2023). Codet5+: Open code large language models for code understanding and generation. arXiv preprint arXiv:2305.07922.

Wang, Y., Wang, W., Joty, S., & Hoi, S. C. H. (2021). CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. arXiv:2109.00859.

Wang, H., Xia, X., Lo, D., He, Q., Wang, X., & Grundy, J. (2021). Context-aware retrieval-based deep commit message generation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, *30*(4), 1–30.

Xing, Z., & Stroulia, E. (2005). UMLDiff: an algorithm for object-oriented design differencing. In *Proceedings of the 20th IEEE/ACM international conference on automated software engineering* (pp. 54–65).

Xu, S., Yao, Y., Xu, F., Gu, T., Tong, H., & Lu, J. (2019). Commit message generation for source code changes. In *IJCAI*.

Yao, Z., Xu, F. F., Yin, P., Sun, H., & Neubig, G. (2021). Learning structural edits via incremental tree transformations. arXiv preprint arXiv:2101.12087.

Yin, P., Neubig, G., Allamanis, M., Brockschmidt, M., & Gaunt, A. L. (2018). Learning to represent edits. arXiv preprint arXiv:1810.13337.

Yu, D., Wang, L., Chen, X., & Chen, J. (2021). Using BiLSTM with attention mechanism to automatically detect self-admitted technical debt. *Frontiers of Computer Science*, *15*(4), Article 154208.

Zeng, Z., Zhang, Y., Zhang, H., & Zhang, L. (2021). Deep just-in-time defect prediction: how far are we? In *Proceedings of the 30th ACM SIGSOFT international symposium on software testing and analysis* (pp. 427–438).

Zheng, W., Shen, T., & Chen, X. (2021). Just-in-time defect prediction technology based on interpretability technology. In *2021 8th international conference on dependable systems and their applications* (pp. 78–89). IEEE.

Zhou, J., Pacheco, M., Wan, Z., Xia, X., Lo, D., Wang, Y., et al. (2021). Finding a needle in a haystack: Automated mining of silent vulnerability fixes. In *2021 36th IEEE/ACM international conference on automated software engineering* (pp. 705–716). IEEE.

Zou, X., Hu, Y., Tian, Z., & Shen, K. (2019). Logistic regression model optimization and case analysis. In *2019 IEEE 7th international conference on computer science and network technology* (pp. 135–139). IEEE.