Information Sciences 697 (2025) 121753

Contents lists available at ScienceDirect

Information Sciences

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

# Move method refactoring recommendation based on deep learning and LLM-generated information

Yang Zhang [a,*], Yanlei Li [a], Grant Meredith [b], Kun Zheng [a], Xiaobin Li [a]

[a] School of Information Science and Engineering, Hebei University of Science and Technology, Shijiazhuang, Hebei, China
[b] Global Professional School, Federation University, Ballarat, Australia

ARTICLE INFO

*Keywords:*
Move method
Refactoring recommendation
Deep learning
Large language models
Prompt engineering

ABSTRACT

Move method refactoring is a prevalent technique typically applied when a method relies more on members of other classes than on its original class. Existing approaches for move method refactoring recommendations have improved accuracy based on deep learning. However, it is challenging to capture the deep semantics behind the code and the true intention of the developer. Furthermore, the accuracy of move method refactoring needs to be improved. To alleviate these problems, this paper proposes MoveRec, a move method refactoring recommendation based on deep learning and LLM-generated information. To generate the dataset, MoveRec selects 58 real-world projects from which it extracts metric, textual, and semantic features. Metric features are derived using static analysis tools. Textual features are generated with LLM to obtain code summaries, and the pre-trained model is used to produce word vectors. Semantic features are obtained by calculating the similarity between the original and target classes. Finally, we construct a dataset with 12,475 samples. A deep learning model CNN-LSTM-GRU is designed for refactoring recommendations. We evaluate MoveRec on this dataset and experimental results show that the average F1 is 74%. Compared to existing methods including PathMove, JDeodorant, JMove, and RMove, MoveRec improves F1 ranging from 9.4% to 53.4%, demonstrating its effectiveness.

## 1. Introduction

The move method is a typical refactoring approach for correcting misplaced functions or methods within a class, which is beneficial for resolving code smells related to “feature envy” [19]. The feature envy code smell manifests as a method excessively relying on methods in other classes, thereby degrading code quality and significantly impacting the maintainability of the software system. Fowler et al. [11] resolved feature envy by suggesting that when a function is found to be overly dependent on functions in other modules, the refactoring operation of the move method should be employed to eliminate design flaws in the code, which is crucial for improving development efficiency and enhancing code quality.

To eliminate the design flaw of feature envy, researchers have proposed numerous refactoring recommendation methods, including JDeodorant [33], JMove [32] and PathMove [15]. These methods generally fall into two defined strategies: heuristic-based approaches, and machine learning-based approaches. Heuristic-based approaches often rely on code metrics or rules defined by developers, resulting in low consistency due to differences in metric selection and expert judgment on thresholds. Machine learning-based

* Corresponding author.
*E-mail addresses:* zhangyang@hebust.edu.cn (Y. Zhang), stone_leey@stu.hebust.edu.cn (Y. Li), g.meredith@federation.edu.au (G. Meredith), zhengkun@hebust.edu.cn (K. Zheng), lixiaobin@buaa.edu.cn (X. Li).

https://doi.org/10.1016/j.ins.2024.121753
Received 10 October 2024; Received in revised form 18 November 2024; Accepted 13 December 2024

Available online 16 December 2024

approaches learn from features extracted from source code to make refactoring recommendations. However, these methods struggle to capture the nuances between different codes of information. To address this, Cui et al. [6] proposed an approach using both metrics and machine learning, leveraging graph embedding and code embedding techniques to more comprehensively capture code information by learning the structural and semantic representations of code snippets. Ujihara et al. [35] introduced a refactoring tool called C-JRefRec, which uses term frequency-inverse document frequency vectors to calculate semantic similarity between functions and target classes for detecting feature envy. Bavota et al. [3] proposed a refactoring tool called MethodBook, which employs a relational topic model method that considers method calls and textual information, including identifiers and comments, to detect feature envy. With the widespread use of LLM, many researchers began to explore their use in refactoring recommendations. For example, POMIAN et al. [38] proposed a method that integrates LLM with static analysis tools to enhance refactoring recommendations, which is implemented as a plugin in IDEs. Further more, DePalma et al. [8] investigated ChatGPT's code refactoring capabilities, demonstrating its advantages in refactoring but also noting issues with hallucinations in results, necessitating human involvement to ensure the validity of outcomes.

Despite the progress made by existing refactoring recommendations for move method, four key challenges remain to be addressed. Firstly, these methods pay insufficient attention to semantic information and struggle to capture effective semantic information. Secondly, there is a lack of publicly available datasets for refactoring recommendations that contain rich code-related information. Thirdly, existing methods suffer from low accuracy, indicating much room for further improvement. Fourthly, while LLM has made breakthroughs in various fields, their application in the refactoring field still needs to be explored.

To address these challenges, this paper proposes "MoveRec", a move method refactoring recommendation based on deep learning and LLM-generated information. To generate the dataset, MoveRec selects 58 real-world projects from which it extracts metric, textual, and semantic features. Metric features are derived using static analysis tools. As a result textual features are generated with LLM to obtain code summaries, and the pre-trained model is used to produce word vectors. Semantic features are obtained by calculating the similarity between the original and target classes. Finally, we construct a dataset with 12,475 samples and use the CNN-LSTM-GRU model designed for refactoring recommendations. We evaluate MoveRec on this dataset and experimental results show that the average F1 is 74%. Compared to existing methods including PathMove, JDeodorant, JMove and RMove, MoveRec improves F1 ranging from 9.4% to 53.4%, demonstrating its effectiveness.

The contribution of this paper can be summarized as follows.

- We construct a dataset for move method refactoring recommendations by extracting metric, textual, and semantic features from 58 real-world projects.
- We propose a refactoring recommendation approach based on deep learning and LLM-generated information incorporating a design of a three-branch deep learning model CNN-LSTM-GRU that significantly enhances the accuracy of refactoring recommendations.
- To demonstrate the effectiveness of MoveRec, we compare its results to existing methods for move method refactoring, proving that it improves 9.4%-53.4% in F1.

The structure of this paper is organized as follows. Section 2 illustrates our motivation. The related works are examined in Section 3. Section 4 presents move method refactoring recommendation based on deep learning and LLM-generated information. Section 5 evaluates the proposed method. Section 6 concludes this work.

## 2. Motivation

To demonstrate our underlying motivation we leveraged RMove [6] to recommend move method refactoring for project JMeter [1]. RMove constructs a dataset using the triplet <movable method, class of the method, target class of the method> and leverages code embedding and graph embedding techniques to capture structural and semantic information in the code, which is then trained through a deep learning model. From the dataset built by RMove, we found that it is insufficient to merely including path information during refactoring. For example, we presented an example including classes *Order* and *Customer* in Fig. 1. Class *Order* contains a method *processOrder()* that calls the method *sendConfirmationEmail()* of class *Customer*. Suppose that RMove relies solely on path information to recommend refactoring. In that case, it may incorrectly recommend moving method *sendConfirmationEmail()* from class *Customer* to class *Order* because class *Order* invokes this method. However, this recommendation is unreasonable because method *sendConfirmationEmail()* is more relevant to the responsibilities of class *Customer*. It involves sending emails, which is a core function of class *Customer*. If we analyze the deep semantics of the code and understand that method *sendConfirmationEmail()* belongs to class *Customer*, it will reduce the occurrence of erroneous refactoring recommendations.

RMove tries to capture the deep semantics behind the code and developers' true intentions when extracting features at different levels, resulting in less accurate refactoring recommendations. Therefore, we attempted to improve the accuracy of the recommendations by exploring deeper semantics behind the code and understanding the real intentions of developers. To this end, we added code-related semantic information based on the original triplet relationship of RMove. To test our new method we compared the performance of the original RMove to the improved RMove. Tabel 1 shows the comparison results in Precision, Recall, and F1. The experimental results show that the Precision, Recall and F1 of the improved RMove were increased by 31.10%( = 82.71-51.61%), 31.09%( = 82.65%-51.56%) and 31.83%( = 82.62%-50.79%), respectively. The accuracy of refactoring recommendations after adding semantic information was significantly improved when compared with the original RMove.

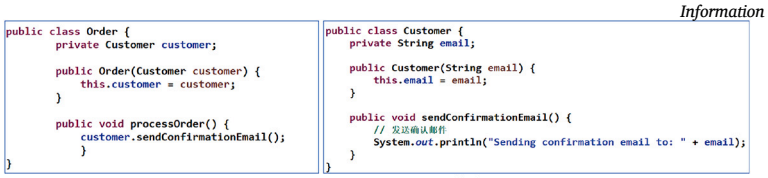


**Fig. 1.** Example.

**Table 1**
Motivation.

| Approach | Precision(%) | Recall(%) | F1(%) |
|---|---|---|---|
| Original RMove | 51.61 | 51.56 | 50.79 |
| Improved RMove | 82.71 | 82.65 | 82.62 |

## 3. Related works

Related research for move method refactoring recommendations can be categorized into heuristic-based, traditional machine learning-based, and deep learning-based, and deep learning-based approaches.

Heuristic-based approaches rely on metrics or thresholds to identify specific code smells and complete refactoring recommendations. Numerous heuristic tools exist, such as JDeodorant, which can detect various code smells including feature envy, long method, and god class, and provide appropriate refactoring recommendations to eliminate these code smells. To ensure the reliability of the suggestions, a condition for recommendation satisfaction, such as compilation conditions, is introduced to ensure that the modified code can be correctly compiled. Terra and colleagues [32] proposed a move method based on the similarity between dependency sets for refactoring recommendations, known as JMove, which compares the dependency relationships established by methods with those potentially established in target classes. Ujihara et al. [35] introduced a refactoring recommendation tool based on semantics and static analysis, which calculates cosine similarity between vectors to obtain semantic similarity between methods and classes. Bavota and colleagues [3] proposed a method using a relational topic model to identify opportunities for the move method. Palomba et al. [24] presented a method called TACO for detecting code smells, based on information retrieval, which can detect code smells such as feature envy and long method.

To address the limitations of heuristic-based methods, various machine learning algorithms have been applied to research related to code smells like feature envy. Barbez et al. [2] proposed a machine learning-based method for detecting code smells such as feature envy and god class. This method collects relevant metrics from multiple code smell detection tools and uses them to train machine learning classifiers. Cui et al. [6] proposed a combined approach using metrics and machine learning, leveraging graph embedding and code embedding techniques to obtain feature representations from code snippets and construct classifiers for refactoring recommendations. With the advancements in machine learning and LLM, using AI tools to automate or assist in coding has become an indispensable productivity tool. Zhang et al. [42] conducted an empirical study collecting all data from Stack Overflow and GitHub, providing empirical foundations for the improvement of AI coding tools through the analysis of the best practices, limitations, and challenges faced by software developers using GitHub Copilot. Cui et al. [7] proposed HMove, a novel hypergraph learning based method for opportunity suggestions. The hypergraph neural network alongside a large language model (LLM) is used to build a refactoring suggestion system.

More recently, research on feature envy has increasingly employed deep learning. Although deep learning models are complex, they can more effectively capture useful information from extracted features. For example. Liu et al. [20] proposed a deep learning-based approach to detect code smells such as feature envy and long method. To identify feature envy, they extracted textual information from the source code and then employed Word2Vec [23] to map it to continuous distributed vectors. Liu and colleagues also calculated the distance between the method and the target class, as well as the distance between the method and its original class. These values were inputted into a Convolutional Neural Network to learn the semantic relationships between identifiers, thereby determining where the method should be placed. Hadj-Kacem et al. [12] proposed a deep learning-based approach to detect code smells, including feature envy and long method. Their approach parsed the source code into an abstract syntax tree and used a variational autoencoder to generate its representation. They further implemented a linear regression classifier to detect feature envy. Sharma et al. [28] systematically compared the detection of code smells and find that recurrent neural networks perform the best. Wang et al. [36] investigated the dimensionality reduction ability of auto-encoder, and thus contributed to the success of deep learning. Li et al. [18] focused on the analysis of the characteristics and mathematical theory of BP neural network and also outlined the shortcomings of BP algorithm as well as several methods for improvement. Liu et al. [22] proposed KANs as promising alternatives to MLPs, opening opportunities for further improving today's deep learning models which rely heavily on MLPs. It is also important to note that feature extraction plays a crucial role in improving the performance of deep learning models. For example, Yu et al. [40] proposed a novel ranking model based on the learning in which visual features and click features are simultaneously utilized to obtain the ranking model.

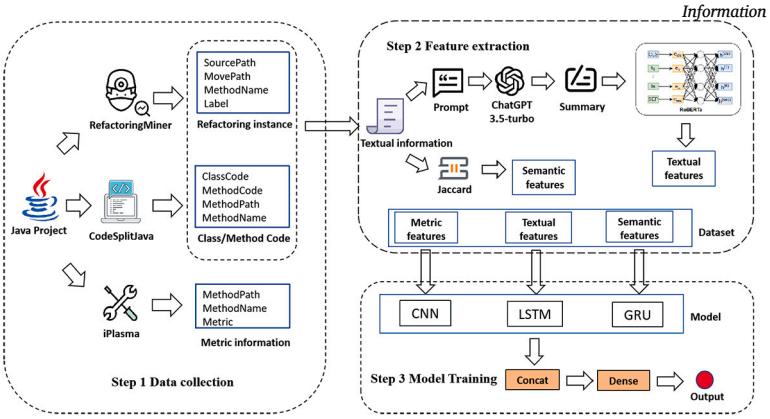


**Fig. 2.** Move method refactoring recommendation based on deep learning and LLM-generated information.

## 4. Design

This section first presents an overview of MoveRec and then introduces the data collection process. Section 4.3 discusses feature extraction under different levels of information. Section 4.4 describes the process of using the extracted features for model training.

### *4.1. Overview*

The framework of MoveRec is shown in Fig. 2. Firstly, the following projects were processed to obtain textual, semantic, and metric information by RefactoringMiner [34], CodeSplitJava [27], and iPlasma [5], respectively. Secondly, we extracted features from the textual information in the dataset based on an LLM and similarity calculations. On one hand, we fine-tuned the LLM by providing prompts and code examples to generate code summaries, and we obtained textual features using the pre-trained model RoBERTa. On the other hand, we calculated the similarity between the path changes from the method's original class to the potential target class and the code summaries to obtain semantic features. Thirdly, we utilized metric features, textual features, and semantic features as distinct inputs for our deep learning model. Specifically, we designed a three-branch CNN-LSTM-GRU model architecture where the CNN branch processed metric features, the LSTM branch handles textual features, and the GRU branch deals with semantic features. After model training, we concatenated the feature vectors and passed them through fully connected layers to complete the classification for refactoring recommendations.

### *4.2. Data collection*

This section describes the process of data collection.

#### *4.2.1. Project selection*

A collection of 58 open-source real-world projects were selected as the source of the dataset. These projects and their configurations are shown in Table 2. NOC, NOM, and LOC represent the number of classes, methods, and lines of code, respectively. The selected projects comprise over 5.65 million LOC, including 52 projects with over 10,000 LOC, such as PMD, Cayenne, and Pinpoint. These projects encompass a total of 74,643 classes and 639,010 methods. Among them, 54 projects have more than 100 classes and 1,000 methods, indicating that these projects are of substantial size. These projects originate from various domains, including static analysis, project management, and performance testing.

#### *4.2.2. Refactoring instances*

A total of 58 real-world applications were selected to construct the dataset. We utilized RefactoringMiner to extract refactoring instances that undergo move method operations, including potentially movable method names, the paths where the method names are located, and the target paths where they can be moved. Simultaneously, we used CodeSplitJava to obtain class-level and method-level code and extract method names and the paths where the methods are located. To aid with integrating the information extracted by both tools, we matched and merged the content extracted through path matching. The merged structure is organized as <Project, SourcePath, MovePath, MethodName, MethodCode, ClassCode, Label>. A screenshot of the results obtained after extracting refactoring instances using the two tools is shown in Fig. 3.

### *4.3. Feature extraction*

To capture deeper semantics and understand the developers' true intentions, we considered three approaches. The first approach was to use information from GitHub history commits. However, we found that these commits were highly subjective and that some programmers may have misunderstandings or missing information when recording commit messages. The second approach was to

**Table 2**
Projects and their configuration.

| Project | Description | NOC | NOM | LOC |
|---|---|---|---|---|
| PMD | Static analysis tool for Java | 1905 | 6382 | 33933 |
| Cayenne | Database access | 2756 | 16598 | 127577 |
| Pinpoint | APM monitoring tool | 2843 | 28753 | 198952 |
| Jenkins | Automated build tools | 660 | 4840 | 33328 |
| Abdera | Parse feed record documents | 668 | 7513 | 48429 |
| Ant | Command-line tools | 882 | 6080 | 42590 |
| FreeMind-MMX | Mind mapping | 532 | 7303 | 65866 |
| JMeter | Performance and load testing procedures | 539 | 5099 | 38080 |
| JTOpen | IBM Toolbox | 2298 | 29187 | 355281 |
| Maven | Software project management | 616 | 4365 | 20505 |
| Weka | Machine Learning Toolbox | 2889 | 37713 | 522439 |
| Accumulo | Distributed key-value storage system | 839 | 19692 | 163471 |
| DrJava | Interactive programming and teaching programs | 1135 | 16738 | 145745 |
| Antlr4 | Grammar analyzer generator | 442 | 3282 | 30987 |
| ArgoUML | UML modeling tools | 1931 | 17129 | 157611 |
| AspectJ | Java extensions | 3280 | 25621 | 217305 |
| Batik | Image rendering | 1684 | 16115 | 165045 |
| Checkstyle | Code review tools | 2874 | 41363 | 742236 |
| Collections | Java data processing tools | 1198 | 5198 | 29729 |
| Displaytag | Build a web user interface | 251 | 911 | 7786 |
| Xalan | Data transformation | 964 | 10359 | 188637 |
| Xerces | Java library for parsing XML | 838 | 10717 | 142249 |
| Roller | Multi-user blog server | 415 | 3084 | 23001 |
| React-Native | Application frameworks | 288 | 3886 | 25515 |
| Design-patterns | Java design pattern | 970 | 2328 | 13567 |
| Mall | Management system | 417 | 3173 | 8896 |
| RxJava | Monitor thread-safe programs | 736 | 4148 | 40666 |
| Elasticsearch | Log analysis tools | 4157 | 34679 | 220249 |
| Retrofit | HTTP client library | 151 | 683 | 6013 |
| Dubbo | Remote calling framework | 246 | 10368 | 63554 |
| JADX | Java code tools | 1023 | 4135 | 23690 |
| MPAndroidChart | Open-source icon library | 162 | 1349 | 12241 |
| DBeaver | Manage and query databases | 2474 | 17708 | 96951 |
| Tutorials | Collection of Java tutorials | 7123 | 30326 | 167263 |
| Lottie | Animated rendering | 143 | 789 | 5372 |
| Glide | Cache library | 474 | 3585 | 27811 |
| Arthas | Java Diagnostic Tools | 614 | 4888 | 36809 |
| AndroidUtilCode | Android open-source tool library | 198 | 2088 | 14374 |
| Netty | Asynchronous event framework | 2653 | 30280 | 254027 |
| ZXing | Processing library | 401 | 2247 | 24958 |
| Halo | Personal blog system | 189 | 938 | 5466 |
| Apollo | Configure distributed applications | 452 | 2512 | 16020 |
| Selenium | Web testing tool | 390 | 2307 | 15589 |
| Nacos | Service management platform | 1465 | 7586 | 47696 |
| Hutool | Java toolkit | 964 | 7750 | 48341 |
| Druid | Database management | 5089 | 29534 | 324052 |
| Termux | Terminal simulator | 151 | 1334 | 14248 |
| Canal | Database synchronization tool | 407 | 3593 | 25867 |
| NewPipe | Android app | 171 | 1107 | 7525 |
| Kafka | Distributed stream processing platform | 1779 | 9966 | 52196 |
| FASTJSON | Processing JSON format data | 3009 | 16403 | 130033 |
| Seata | Distributed transaction solutions | 1144 | 15806 | 92980 |
| Signal-Android | Instant messaging app | 1084 | 8245 | 51566 |
| Hystrix | Control distributed system interaction | 381 | 3483 | 32764 |
| SkyWalking | Performance monitoring tool | 922 | 2539 | 18319 |
| Gson | JSON format conversion tool | 147 | 1103 | 10130 |
| LibGDX | Game development framework | 2230 | 44102 | 218981 |
| Total | | **74643** | **639010** | **5654511** |

use traditional tools to obtain information from historical commits as semantic supplements. However, traditional tools are relatively singular in nature and rely heavily on developers' habits of annotating modified information, which is still insufficient for semantic supplementation. The third approach was to use an LLM, by applying prompt learning [14] combined with code snippets to generate code summaries. After comparing the above three methods, we chose to use LLM to assist in completing deeper semantic information supplementation. Firstly, an LLM performs well in code understanding and code summarization. Secondly, previous work has not applied LLM to move method refactoring recommendation research.

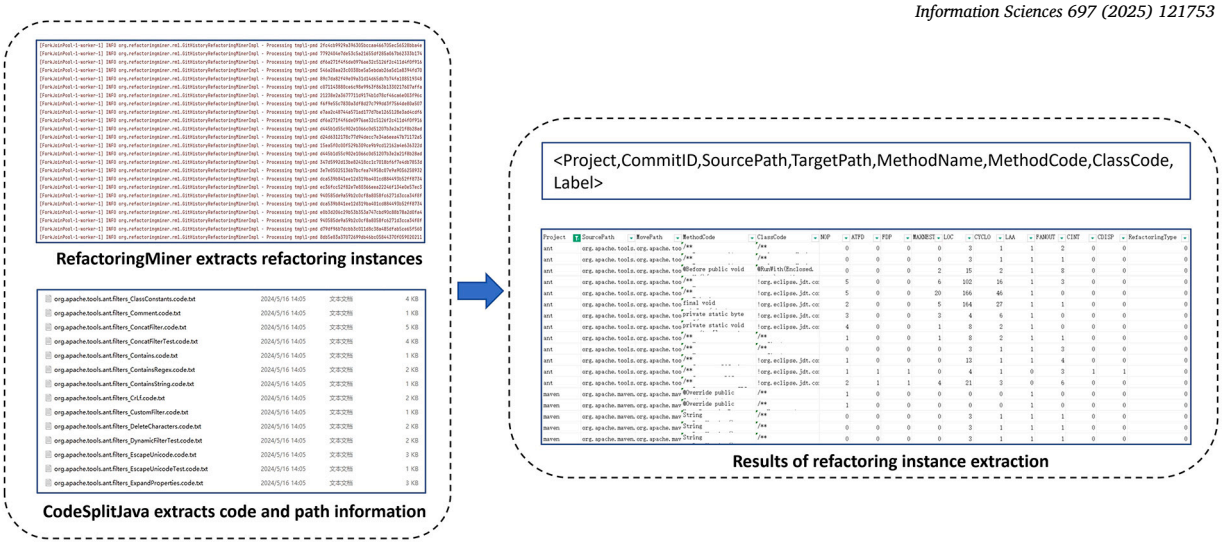


**Fig. 3.** The results of extracting refactoring instances.

### *4.3.1. Design of prompts*

Designing the prompt template was a key step to ensure the accuracy and effectiveness of the generated information. We designed five prompt templates from the perspectives of clear instructions, role-playing, and word count limits [25]. $p_1$ and $p_2$ specify instructions, $p_3$ and $p_4$ add word count limits based on clear instructions, and $p_5$ adds the role of an expert software refactoring engineer with optimized instructions based on clear instructions and word count limits.

$p_1$ is designed to provide clear and comprehensive instructions for summarizing the class-level code structure, functionality, and features. It emphasizes the need to cover key aspects such as class name, inheritance relationship, functional description, member variables, method collection, primary functionality, and key features, all within a single sentence.

> $p_1$: Summarize the class-level code structure, functionality, and features in a single sentence, covering aspects such as class name, inheritance relationship, functional description, member variables, method collection, primary functionality, key features. + [CODE]

$p_2$ follows a similar approach to $p_1$ but focuses on summarizing method-level code functionality. It instructs the user to cover aspects such as method name, input parameters, output results, method functionality, algorithmic logic, exception handling, boundary cases, and performance considerations.

> $p_2$: Summarize the method-level code functionality, covering aspects such as method name, input parameters, output results, method functionality, algorithmic logic, exception handling, boundary cases, and performance considerations. + [CODE]

$p_3$ builds upon the clear instructions provided in $p_1$ by adding a word count limit. This template encourages a concise summary while still covering all the essential aspects of the class-level code.

> $p_3$: Summarize the class-level code structure, functionality, and features in a single sentence, covering aspects such as class name, inheritance relationship, functional description, member variables, method collection, primary functionality, and key features. Please make a concise summary. + [CODE]

$p_4$ mirrors the structure of $p_3$ but applies it to method-level code functionality. It combines clear instructions with a word count limit to ensure a concise yet comprehensive summary.

> $p_4$: Summarize the method-level code functionality, covering aspects such as method name, input parameters, output results, method functionality, algorithmic logic, exception handling, boundary cases, and performance considerations. Please make a concise summary. + [CODE]

$p_5$ introduces a role-playing element by assigning the user's role as an expert in software refactoring field. This template combines optimized instructions based on clear instructions and word count limits, encouraging a concise summary while leveraging the expertise of the role.

$p_5$: You are a software engineer who is proficient in refactoring. You will do the work of code summary. Please make a concise summary based on the following code features. + [CODE]

We compared different prompts in 10 Java projects. $p_1$ and $p_2$ describe class-level and method-level information, respectively. Through the generated code summaries, it can be seen that they can describe the functional information of the code. However, the information is too long and cannot be summarized effectively. To this end, we leveraged $p_3$ and $p_4$ which added prompts for concise summaries based on $p_1$ and $p_2$. It was found that $p_4$ generated more concise summaries while correctly describing the code functions compared to $p_1$, $p_2$, and $p_3$, which indicates that adding word count limits in prompts is effective. Finally, we constructed the role of an expert software engineer in prompt template $p_5$, clearly specifying the instruction to perform code summarization and limiting the word count of the code summary. The code summaries generated by $p_5$ accurately describe the code functions, are concise, and can generate code summaries for different granularity levels using a single prompt template. Therefore, we chose $p_5$ as our prompt template for LLM generation of textual information.

#### 4.3.2. Generation of textual information with LLM

The generation of textual information with LLM is shown in Algorithm 1. Firstly, we used the method level code and class level code from the collected data samples as inputs. To ensure that LLM can generate high-quality textual information, we needed to perform token checks on the input code. Due to the upper limit of input token for LLM, if the token of the input code exceeds this limit, we divided the code into multiple modules and input them into the model in sequence. After that, we integrated the prompt with the code module to be analyzed through prompt engineering, subsequently invoking the LLM for comprehensive analysis. Finally, upon receiving the prompt and code module, LLM generated corresponding code summary. In the algorithm, we set Max_Token limit to ensure that the input code conforms to the constraints of the LLM used. The algorithm is described as follows.

1. Traverse the method level code in the data sample, where $N$ (line 1) is the number of data samples.
2. For each method-level code $C_{\text{method}}$ in the data sample, if the input of $C_{\text{method}}$ is greater than Max_Token (line 2), split $C_{\text{method}}$ into modules (line 3). Otherwise, set modules to $C_{\text{method}}$ (line 4).
3. For each module in modules, generate the response $S_{\text{method}}$ using the prompt and messages (lines 5 to 6).
4. Traverse the class-level code in the data sample (line 7).
5. For each class level code $C_{\text{class}}$ in the data sample, if the input of $C_{\text{class}}$ is greater than Max_Token (line 8), split $C_{\text{class}}$ into modules (line 9). Otherwise, set modules to $C_{\text{class}}$ (line 10).
6. For each module in modules, generate the response $S_{\text{class}}$ using the prompt and messages (lines 11 to 12).

**Algorithm 1:** Generation of textual information with LLM.

**Input:** $C_{\text{method}}, C_{\text{class}}$
**Output:** $S_{\text{method}}, S_{\text{class}}$
1 **for** $n = 1$ **to** $N$ *in* $C_{method}$ **do**
2  **if** $Len(C_{method}) > Max\_Token$ **then**
3  modules ← split_code_into_modules($C_{\text{method}}$)
4  **else**
5  modules ← $C_{\text{method}}$
6  **for** *module in modules* **do**
7  $S_{\text{method}}$ ← response ← prompt ← messages
8 **for** $n = 1$ **to** $N$ *in* $C_{class}$ **do**
9  **if** $Len(C_{class}) > Max\_Token$ **then**
10  modules ← split_code_into_modules($C_{\text{class}}$)
11  **else**
12  modules ← $C_{\text{class}}$
13  **for** *module in modules* **do**
14  $S_{\text{class}}$ ← response ← prompt ← messages

During the generation process, we controlled the parameters of the LLM. The specific parameter settings are shown in Table 3. The temperature controls the randomness or creativity of the model's generated textual information, with a parameter value ranging from 0 to 2. When the temperature is set lower, the model tends to choose words with higher probabilities, resulting in more deterministic and consistent textual information. To ensure the accuracy and consistency of the generated textual information, the temperature parameter value is set to 0. top_p is a sampling strategy that limits the range of the probability distribution considered by the model when generating the next word, and it works together with temperature to determine the randomness of the generated textual information. Presence_penalty controls the degree of topic engagement; the higher the value, the more likely it is to discuss new topics. To ensure the consistency of the topic, the parameter value is set to 0. Frequency_penalty is used to reduce the repetition of generated words. When generating code summaries, different code modules in a project often have certain connections, leading to the repetition of generated words. Therefore, the parameter value is set to 0.

Fig. 4 illustrates the process of generating textual information with LLM, as well as the results of generating code summaries at the method and class levels. The designed prompts are applied to the LLM and then generate code summaries at both method and class levels.

**Table 3**
Parameter configuration of LLM.

| Parameter | Description | Value |
|---|---|---|
| temperature | 0-2, higher values are more random | 0 |
| top_p | 0-2, higher values are more diverse | 1 |
| presence_penalty | Topic repetition | 0 |
| frequency_penalty | Repetition penalty factor | 0 |

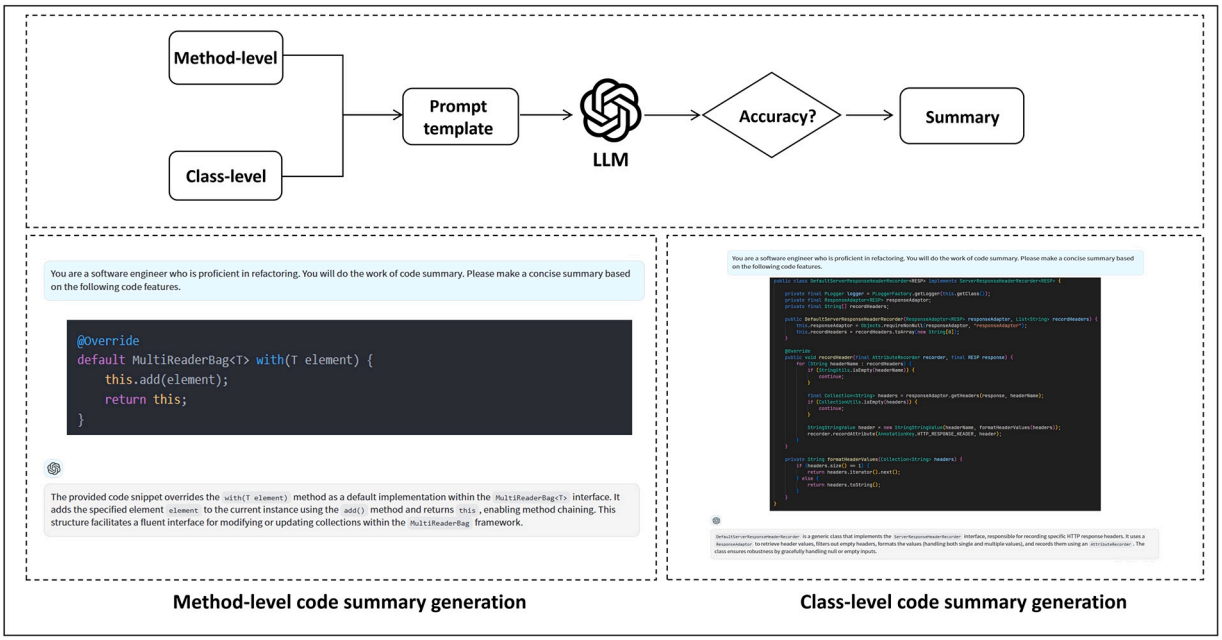


**Fig. 4.** Generating textual information by LLM.

### *4.3.3. Representation of textual information*

We chose LSTM and GRU models to represent textual information, as both are neural network models specifically designed for processing sequential data. In the context of move method refactoring recommendations, the contextual information and temporal dependencies of code fragments are crucial. LSTM and GRU effectively capture these dependencies, thereby enhancing the accuracy of the model.

The use of a LLM as an auxiliary tool is motivated by several factors. Firstly, traditional methods for extracting information from method-level code yield relatively limited insights. In contrast, leveraging a LLM can provide a more comprehensive understanding of the code. Secondly, class-level code often encompasses numerous method functions, and traditional methods struggle to capture the developers' true intentions and the deeper meanings embedded within the code. By employing a LLM, it becomes possible to summarize and abstract class-level code, thereby uncovering deeper characteristics and insights through a more nuanced comprehension of the code.

Most existing works primarily employ word embedding models such as Word2Vec and BERT [9]. While these models consider contextual information to some extent, they still fall short in capturing and representing semantic information compared to more advanced pre-trained models. RoBERTa is an optimized pre-trained model based on BERT, featuring a LLM's parameters and the ability to train on more data. It can capture rich semantic and contextual information, thereby enhancing prediction performance in subsequent model training processes.

In Fig. 5, the RoBERTa model is applied to map the textual descriptions of method-level and class-level code into fixed word vectors, which reflects the semantic relevance between codes through the generated vectors. The word vectors mapped by the RoBERTa model are used as inputs for the long short term memory recurrent neural network (LSTM) layer, as shown in Fig. 5 for the representation of textual features by LSTM. LSTM introduces the input gate, forget gate, output gate, and cell state. The input gate determines the extent to which the input data at the current time step updates the cell state, controlling the amount of update for each cell state; the forget gate decides which information from the previous time step's cell state should be forgotten, controlling which parts of the previous cell state are retained or discarded; the output gate determines how the current time step's cell state is mapped to the hidden state, deciding which parts of the cell state will be output. The cell state contains the information that the network needs to remember at the current time step, and at each time step, the cell state is updated based on the current input and adjustments from the past state. The calculation process is shown in Equation (1):

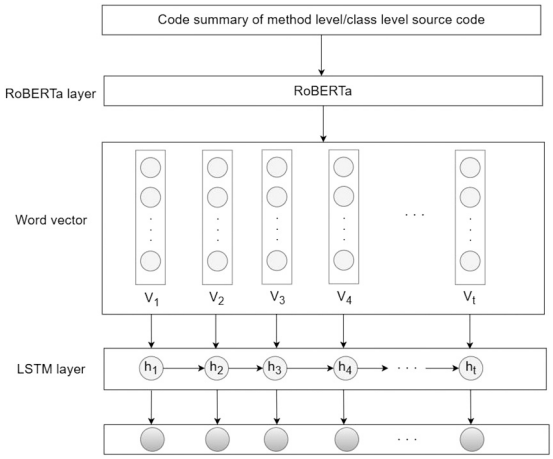


**Fig. 5.** Representation textual features by LSTM.

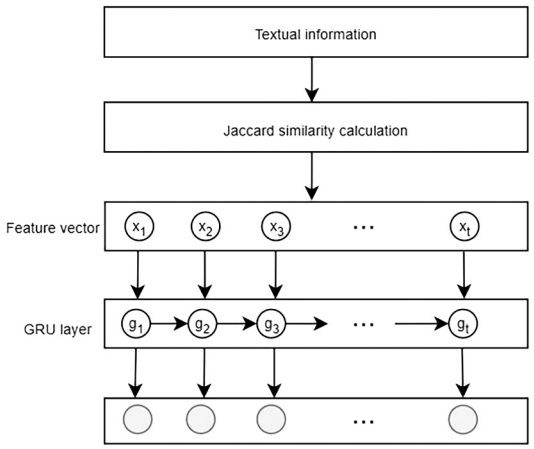


**Fig. 6.** Representation textual features by GRU.

$$\left.\begin{aligned} i_t &= \sigma(W \cdot [h_{t-1}, V_t] + b) \\ f_t &= \sigma(W \cdot [h_{t-1}, V_t] + b) \\ o_t &= \sigma(W \cdot [h_{t-1}, V_t] + b) \\ \tilde{C}_t &= \tanh(W \cdot [h_{t-1}, V_t] + b) \\ C_t &= f_t \cdot C_{t-1} + i_t \cdot \tilde{C}_t \\ h_t &= o_t \cdot \tanh(C_t) \end{aligned}\right\} \tag{1}$$

Where $i_t$, $f_t$, $o_t$ represent the input gate, forget gate, and output gate, respectively. $\sigma$, $W$, tanh, $V_t$, $b$, $h_t$, $h_{t-1}$ represent the activation function, weight matrices, activation function, input feature vector, trainable parameters, current cell output, and previous cell output, respectively. $C_{t-1}$, $\tilde{C}_t$, $C_t$ represent the previous cell state, candidate cell state for the current time step, and current cell state, respectively.

Textual information includes code summaries at the method and class levels, as well as path changes before and after refactoring. To capture more semantic information, we separately calculate the similarity of code summaries at the method and class levels, and the path changes before and after refactoring to obtain feature vectors. During the calculation process, we employed the Jaccard similarity coefficient to calculate textual information similarity and measure distances between objects. The calculated feature vectors are then used as inputs for the gated recurrent unit (GRU), which is represented in the textual features as shown in Fig. 6.

The GRU contains an internal reset gate and an update gate. The reset gate determines which information from the previous time step's hidden state should be ignored, while the update gate decides how much information from the previous time step's hidden state should be retained and passed on to the current time step. By introducing the GRU model to train on the calculated similarity values, we can better capture the semantic relationships contained in the textual information. The calculation process is shown in Equation (2):

**Table 4**
Metrics.

| Metric | Description |
|---|---|
| NOP | Number of Parameters |
| ATFD | Access to Foreign Data |
| FDP | Foreign Data Providers |
| MAXNESTING | Maximum Nesting Level |
| LOC | Lines of Code |
| CYCLO | Cyclomatic Complexity |
| LAA | Locality of Attribute Accesses |
| FANOUT | Fan Out |
| CINT | Coupling Intensity |
| CDISP | Coupling Dispersion |

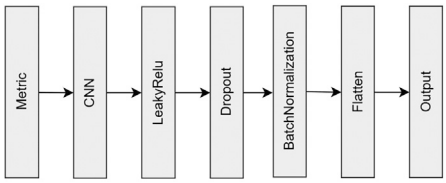


**Fig. 7.** Representation of metric features.

$$\left.\begin{aligned} r_t &= \sigma(W \cdot [g_{t-1}, x_t] + b) \\ z_t &= \sigma(W \cdot [g_{t-1}, x_t] + b) \\ \tilde{g}_t &= \tanh(W \cdot [r_t \cdot g_{t-1}, x_t] + b) \\ g_t &= (1 - z_t) \cdot g_{t-1} + z_t \cdot \tilde{g}_t \end{aligned}\right\} \tag{2}$$

Where $r_t$, $z_t$, $x_t$, tanh, $\sigma$, $W$, $b$ represent the reset gate, update gate, feature vector after similarity computation, tanh activation function, Sigmoid activation function, and weight matrices, the trainable parameters respectively. $g_{t-1}$, $\tilde{g}_t$, $g_t$ represent the parameters for the reset gate, update gate, candidate state, and current state, respectively, where $g_{t-1}$ denotes the state carried over from the previous time step, $\tilde{g}_t$ denotes the state after reset, and $g_t$ denotes the state at the current time step.

#### 4.3.4. Generating metrics information

Using the static analysis tool iPlasma to generate metric information, we selected 10 different metric items that encompassed characteristic information related to move method refactoring. These metrics included: NOP (Number of Parameters), which reflects the complexity and coupling of the method; ATFD (Access to Foreign Data), indicating the number of times the method accesses external classes or global variables; FDP (Foreign Data Providers), representing the number of classes that provide external data to the current class or method; MAXNESTING (Maximum Nesting Level), reflecting the complexity of the code's control flow; LOC (Lines of Code), indicating the size of the method or class; CYCLO (Cyclomatic Complexity), measuring the number of independent paths in the code and reflecting its complexity and testability; LAA (Locality of Attribute Accesses), reflecting the distribution of attribute accesses within the class; FANOUT (Fan-Out), representing the number of methods called by the method; CINT (Coupling Intensity), indicating the strength of dependencies between classes or methods; and CDISP (Coupling Dispersion), reflecting the distribution of dependencies between code modules. These metrics collectively provide a comprehensive view of the code's structural and behavioral characteristics, aiding in the analysis and refactoring of move method.

We selected these metrics for the following reasons. Firstly, these metrics have been widely used in previous research to reflect the structural information of the code. Secondly, these metrics are relevant to determining whether a move method is necessary. The detailed information on these code metrics is shown in Table 4.

#### 4.3.5. Representation of metric features

The metric information obtained using static analysis tools is used as input for the classifier, with this part employing a convolutional neural network(CNN) model to capture deeper features in the metric information. The CNN model consists of an input layer, convolutional layer, activation function layer, dropout layer, batch normalization layer, and flattening layer. The convolution kernel size at the input is 6, with 32 convolution kernels, and the padding method is the same with a stride of 1. LeakyReLU accelerates the convergence speed of gradient descent and improves the stability of the model. The dropout layer randomly discards a certain proportion of neuron outputs to prevent overfitting. Batch normalization helps to address the issues of gradient vanishing and gradient explosion. The flattening layer converts all dimensions of the input data into a one-dimensional vector, facilitating its transmission to the fully connected layer for processing. The representation of metric features is shown in Fig. 7.

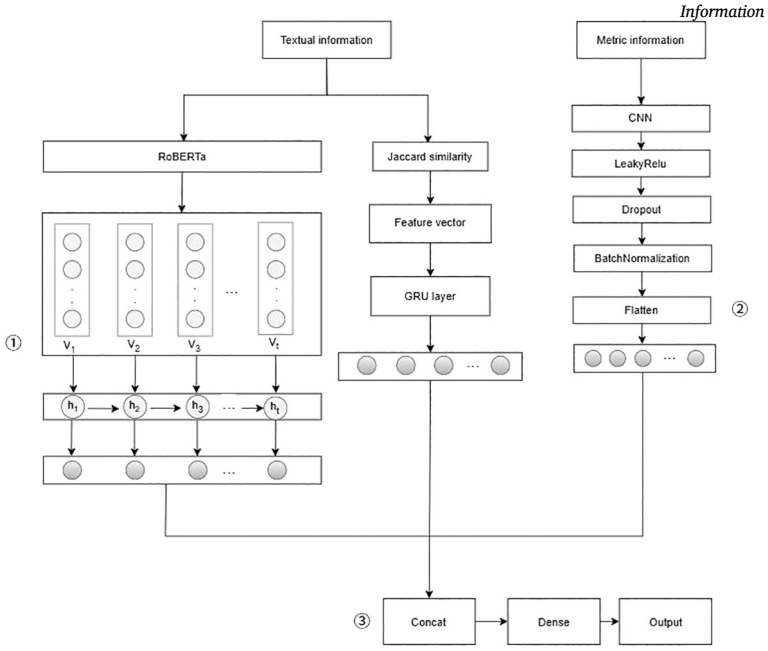


**Fig. 8.** Model of MoveRec.

### *4.4. Model*

Considering that the input of the model comprises textual, semantic, and metric features, we designed a three-branch model structure CNN-LSTM-GRU as shown in Fig. 8. Unlike the single-branch and dual-branch models used in previous works [43], this approach incorporates more features at different levels. Each branch learns the representation of various features, and using a three-branch model has improved the accuracy compared to single-branch and dual-branch models.

In the first part (①), textual information is used as input. The RoBERTa model maps code summaries at the method and class levels in the textual information into word vectors. The Jaccard similarity is used to calculate the similarity values of code summaries and paths before and after refactoring as feature vectors. Then, the GRU and LSTM models are used to fully capture the semantic relationships between codes and paths in the textual information. In the second part (②), the obtained metric information is used as input for the CNN model, which extracts data features from the metric information through the CNN model. In the third part (③), the textual and metric feature representations are concatenated through a merging layer. Finally, the fully connected layer maps to the final output, thereby completing the classification.

## 5. Evaluation

This section first presents 6 research questions and then provides the evaluation metrics. Experimental results are analyzed and threats to validity are discussed.

### *5.1. Research questions*

In the experiments, we evaluate the effectiveness of MoveRec by answering 6 research questions (RQs).

RQ1: How effective is MoveRec in recommending move method refactoring?

RQ2: How effective are RoBERTa, ALBERT, and XLNet in MoveRec?

RQ3: How effectively does textual and metric information generated with an LLM outperform the effect of only including metric information? Compared to textual information obtained from other tools, is the textual information generated with LLM more advantageous?

RQ4: How effective is MoveRec in performance compared to traditional machine learning and deep learning models?

RQ5: How effective is MoveRec in performance compared to existing methods for resolving feature envy in the move method?

RQ6: How about the time cost of each component of MoveRec?

RQ1 focused on the effectiveness of refactoring recommendations for the move method. To answer RQ1, eight Java projects from different domains were selected for cross-validation, and the results of refactoring recommendations for the move method are recorded in terms of precision, recall, and F1. RQ2 focused on which pre-trained model can exhibit the best performance during preprocessing. To answer RQ2, three pre-trained models including RoBERTa, ALBERT, and XLNet were compared. RQ3 focused on whether the textual information generated with ChatGPT3.5-turbo can effectively improve the effect of refactoring recommendations for the move method. To answer RQ3, the tool Pydriller was compared with ChatGPT3.5-turbo. RQ4 focused on whether the designed

**Table 5**
Cross-validation results(%).

| Project | Precision | Recall | F1 |
|---|---|---|---|
| Antlr4 | 91.26 | 98.95 | 94.95 |
| Collections | 84.20 | 80.91 | 79.97 |
| Roller | 87.67 | 81.41 | 79.91 |
| PMD | 89.77 | 96.34 | 92.94 |
| Pinpoint | 92.18 | 99.78 | 95.83 |
| Accumulo | 78.21 | 72.50 | 71.03 |
| Batik | 65.81 | 68.89 | 66.72 |
| AndroidUtilCode | 64.81 | 64.36 | 64.09 |
| Avg | **81.74** | **82.89** | **80.68** |

three-branch model is more effective compared to traditional machine learning and deep learning. To answer RQ4, the three-branch model was compared with traditional machine learning and deep learning models, and the performance of the CNN-LSTM-GRU is analyzed based on precision, recall, and F1 on our dataset. RQ5 focused on whether the method proposed in this paper is superior to other methods for move method refactoring. To answer RQ5, MoveRec was compared with currently existing move method refactoring methods JDeodorant, JMove, PathMove, and RMove in terms of precision, recall, and F1. RQ6 focused on the time performance of MoveRec during the refactoring of move method. For five open-source projects in the dataset, the time taken by MoveRec throughout the entire refactoring process was recorded.

### 5.2. Evaluation metric

The performance of the method was evaluated through metrics such as precision, recall, and F1 on the test set. Precision represents the probability that a sample predicted as positive is correctly predicted as positive among the predicted results. The calculation formula is shown in Eq. (3).

$$Precision = \frac{TP}{TP + FP} \tag{3}$$

Recall represents the probability that a positive sample in the original sample is correctly predicted as positive in the end. The calculation formula is shown in Eq. (4).

$$Recall = \frac{TP}{TP + FN} \tag{4}$$

Where TP and TN represent the positive and negative samples of the sample, respectively. FP represents the number of incorrect samples classified as correct samples, and FN indicates the number of correct samples classified as incorrect samples. F1 is equivalent to the weighted average of precision and recall, with a value range between 0 and 1. The higher the F1, the better the balance is achieved when both precision and recall are at their highest. The calculation formula is shown in Eq. (5).

$$F1 = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{5}$$

### 5.3. Results

This section presents the results of each RQ.

#### 5.3.1. Result of RQ1

To answer RQ1, we conducted experiments on a dataset constructed from 58 projects. Furthermore, we chose eight projects for Leave-One-Out Cross Validation (LOOCV) comparisons to validate the effectiveness of the method. We chose 8 projects out of 58 for LOOCV because the dataset is constructed with a relatively large number of projects and these projects are representative and large-scale.

We divided the dataset constructed from 58 projects into training and testing sets in an 8:2 ratio, and evaluated MoveRec using precision, recall, and F1. The precision, recall, and F1 are 95.66%, 95.50%, and 95.50%.

The cross-validation results are shown in Table 5. It presents the overall performance of MoveRec by eight open-source test projects. The experimental results indicated that MoveRec displayed a strong average performance on the testing program for cross-validation, achieving an average precision, recall, and F1 of 81.74%, 82.89%, and 80.68%, respectively. Additionally, MoveRec attained the highest testing performance on the Pinpoint testing program, with Pinpoint's precision, recall, and F1 reaching 92.18%, 99.78%, and 95.83%, respectively.

For testing programs such as Accumulo, Batik, and AndroidUtilCode, MoveRec's performance showed a significant decline. This may have been due to the large difference in the ratio of positive and negative samples in these actual programs, which caused MoveRec to predict some samples without move method as samples with move method.

**Table 6**
Performance comparison of pre-trained models(%).

| | RoBERTa | | | XLNet | | | ALBERT | | |
|---|---|---|---|---|---|---|---|---|---|
| Model | Precision | Recall | F1 | Precision | Recall | F1 | Precision | Recall | F1 |
| DT | 52.61 | 44.86 | 43.15 | 47.49 | 41.56 | 40.32 | 41.42 | 40.03 | 35.35 |
| NB | 73.70 | 72.79 | 72.96 | 54.80 | 55.08 | 54.62 | 49.45 | 49.67 | 49.20 |
| SVM | 69.10 | 42.44 | 35.62 | 38.44 | 36.96 | 34.39 | 45.85 | 40.66 | 39.30 |
| LR | 81.09 | 81.21 | 81.14 | 70.02 | 70.25 | 70.09 | 70.29 | 70.66 | 70.33 |
| RF | 93.44 | 93.24 | 93.30 | 83.58 | 83.64 | 83.61 | 83.18 | 83.22 | 83.16 |
| XGB | 97.26 | 97.27 | 97.26 | 90.31 | 90.36 | 90.32 | 89.81 | 89.90 | 89.83 |
| CNN | 93.11 | 93.08 | 93.09 | 79.20 | 77.87 | 78.17 | 76.70 | 76.43 | 76.53 |
| LSTM | 60.04 | 55.08 | 54.07 | 56.56 | 57.15 | 56.68 | 61.03 | 55.28 | 49.39 |
| GRU | 68.71 | 50.76 | 45.34 | 64.21 | 63.44 | 63.66 | 65.54 | 64.41 | 63.49 |
| Avg | **76.56** | **70.08** | **68.44** | 64.96 | 64.03 | 63.54 | 64.81 | 63.36 | 61.84 |

### *5.3.2. Result of RQ2*

To answer RQ2, we compared the performance of pre-trained model RoBERTa against ALBERT and XLNet on 6 machine learning and 3 deep learning models including Decision Tree (DT) [29], Naive Bayes (NB) [37], Support Vector Machine (SVM) [31], Logistic Regression (LR) [17], Random Forest (RF) [26], Extreme Gradient Boosting (XGB) [4], CNN [13], LSTM [41], and GRU [10]. The experimental results are shown in Table 6.

Compared to the pre-trained models XLNet [39] and ALBERT [16], RoBERTa [21] exhibited the best performance across nine different classifiers, with average precision, recall and F1 of 76.56%, 70.08% and 68.44%, respectively. This represented an increase of 11.60% (=76.56% - 64.96%) and 11.75% (=76.56% - 64.81%) in average precision, 6.05% (=70.08% - 64.03%) and 6.72% (=70.08% - 63.36%) in average recall, 4.90% (=68.44% - 63.54%) and 6.60% (=68.44% - 61.84%) in average F1, respectively, compared to XLNet and ALBERT.

Among the tested machine learning models, RoBERTa's performance varied significantly. For instance, Decision Tree (DT) and Support Vector Machine (SVM) exhibited relatively poor performance. With precision, recall, and F1 of 52.61%, 44.86%, and 43.15% for DT. 69.10%, 42.44%, and 35.62% for SVM, respectively. These were notably below the average, indicating challenges in these models.

In contrast, RoBERTa showed better performance on Naive Bayes (NB) and Logistic Regression (LR), with precision, recall, and F1 of 73.70%, 72.79%, and 72.96% for NB, 81.09%, 81.21% and 81.14% for LR, respectively. While these results are above the average, the performance advantage was not substantial. However, RoBERTa excels in Random Forest (RF) and XGBoost (XGB), achieving precision, recall, and F1 of 93.44%, 93.24%, and 93.30% for RF, 97.26%, 97.27% and 97.26% for XGB, respectively. These results indicated that the vectors processed by RoBERTa significantly enhanced the performance of RF and XGB.

RoBERTa also demonstrated notable improvements in Convolutional Neural Network (CNN) performance, with precision, recall and F1 of 93.11%, 93.08% and 93.09%, respectively. This represented improvements of 13.91%, 15.21%, and 14.92% in precision, recall, and F1 compared to XLNet, 16.41%, 16.65%, and 16.56% compared to ALBERT. On Long Short-Term Memory (LSTM), RoBERTa achieves precision, recall, and F1 of 60.04%, 55.08% and 54.07%, respectively. RoBERTa showed a 3.48% improvement in precision over XLNet and 4.68% improvement in F1 over ALBERT. On Gated Recurrent Unit (GRU), RoBERTa achieves precision, recall, and F1 of 68.71%, 50.76%, and 45.34%, respectively, with 4.50% improvement in precision over XLNet and 3.17% improvement in precision over ALBERT.

XGB stood out with the highest precision, recall, and F1, indicating levels of high accuracy and stability. RF and CNN also showed strong performance in precision and F1, while SVM struggled with recall and F1. LSTM and GRU exhibited relatively low recall and F1, and DT performed poorly across all metrics.

### *5.3.3. Result of RQ3*

To answer RQ3, we compared the impact of adding textual information versus not adding textual information on the results of refactoring recommendations for the move method. We contrasted the results of classifiers on the move method refactoring dataset with combined textual and metric information inputs versus metric information inputs only. Additionally, we included the comparison tool Pydriller [30], which can obtain detailed information from historical commits, such as the committer, commit time, and commit messages. We used Pydriller to retrieve detailed information from each historical commit as textual information. Then, combined it with metric information extracted by static analysis tools, and tested the differences between ChatGPT3.5-turbo and Pydriller tools on 9 different classifiers to analyze which one performs better. The experimental results are shown in Table 7.

The experimental results demonstrated that the incorporation of both textual and metric information significantly enhanced model performance compared to using metric information alone. Specifically, by combining these inputs we improved the average F1 by 14.98% (=64.01%-49.03%), indicating that the textual information generated with ChatGPT3.5-turbo played a crucial role in enhancing the final results.

Moreover, when comparing the performance of textual information obtained from the Pydriller tool combined with metrics across nine different classifiers, ChatGPT3.5-turbo showed superior performance. ChatGPT3.5-turbo improved the average precision, recall and F1 by 4.34% (=67.29%-62.95%), 6.55% (=65.27%-58.72%) and 5.24% (=64.01%-58.77%) over the Pydriller tool. This indi-

**Table 7**
Performance comparison of different tools and different feature inputs (%).

| Model | ChatGPT3.5-turbo | | | | | | Pydriller | | |
|---|---|---|---|---|---|---|---|---|---|
| | Text and Metric information | | | Metric information | | | Text and Metric information | | |
| | Precision | Recall | F1 | Precision | Recall | F1 | Precision | Recall | F1 |
| DT | 46.45 | 48.20 | 45.43 | 53.06 | 52.10 | 51.18 | 40.73 | 38.99 | 37.01 |
| NB | 62.83 | 45.18 | 38.12 | 52.59 | 39.75 | 29.85 | 54.41 | 59.85 | 56.40 |
| SVM | 35.95 | 37.04 | 35.45 | 20.08 | 32.48 | 16.61 | 24.44 | 26.77 | 15.75 |
| LR | 72.37 | 72.64 | 72.43 | 48.51 | 46.29 | 46.31 | 63.96 | 61.24 | 61.43 |
| RF | 84.56 | 84.66 | 84.59 | 64.94 | 62.79 | 62.24 | 87.72 | 70.95 | 77.44 |
| XGB | 91.62 | 91.66 | 91.62 | 68.17 | 64.35 | 63.83 | 85.77 | 77.61 | 81.26 |
| CNN | 78.80 | 76.94 | 77.16 | 56.58 | 53.94 | 53.69 | 76.15 | 61.89 | 68.17 |
| LSTM | 61.69 | 60.38 | 60.56 | 60.80 | 59.85 | 58.02 | 60.62 | 60.09 | 60.35 |
| GRU | 71.35 | 70.69 | 70.73 | 60.21 | 60.50 | 59.52 | 72.78 | 71.12 | 71.12 |
| Avg | **67.29** | **65.27** | **64.01** | 53.88 | 52.45 | 49.03 | 62.95 | 58.72 | 58.77 |

**Table 8**
Comparison of MoveRec with traditional machine learning and deep learning models (%).

| Model | Precision | Recall | F1 |
|---|---|---|---|
| DT | 46.45 | 48.20 | 45.43 |
| NB | 62.83 | 45.18 | 38.12 |
| SVM | 35.95 | 37.04 | 35.45 |
| LR | 72.37 | 72.64 | 72.43 |
| RF | 84.56 | 84.66 | 84.59 |
| XGB | 91.62 | 91.66 | 91.62 |
| CNN | 78.80 | 76.94 | 77.16 |
| LSTM | 61.69 | 60.38 | 60.56 |
| GRU | 71.35 | 70.69 | 70.73 |
| MoveRec | **95.66** | **95.50** | **95.50** |

cated that ChatGPT3.5-turbo outperformed Pydriller in overall performance. This was likely due to ChatGPT3.5's advanced natural language processing capabilities and its ability to generate more contextually relevant textual information than Pydriller.

Among the individual classifiers, RF and XGB exhibited the most significant improvements when using ChatGPT3.5-turbo to generate textual information. Combined with textual information, RF's precision, recall, and F1 improve by 19.62% (=84.56%-64.94%), 21.87% (=84.66%-62.79%), and 22.35% (=84.59%-62.24%). Similarly, XGB's scores improve by 23.45% (=91.62%-68.17%), 27.31% (=91.66%-64.35%), and 27.79% (=91.62%-63.83%). These improvements suggested that the additional textual information provided by ChatGPT3.5-turbo helped these models to better capture the underlying patterns in the data. In contrast, classifiers showed less pronounced improvements such as DT and SVM. Combined with textual information, DT improved the average precision, recall and F1 by -6.61% (=46.45%-53.06%), -3.90% (=48.20%-52.10%), and -5.75% (=45.43%-51.18%), respectively. This indicated that while the textual information provided some benefit, the inherent limitations of these models may still hinder their performance. Such as overfitting in DT and sensitivity to kernel choice in SVM.

Overall, the results indicate the importance of incorporating high-quality textual information in machine and deep learning models. At the same time, it has been proved that the information generated with LLM is superior to traditional tools Pydriller.

### *5.3.4. Result of RQ4*

To address RQ4, we compared the proposed MoveRec method with traditional machine learning and deep learning models. The experimental results are shown in Table 8. Among the machine learning models, DT and SVM exhibited relatively low performance. NB showed better precision at 62.83%, but its recall and F1 were relatively low at 45.18% and 38.12%, respectively. RF and XGB demonstrated strong performance. RF achieved precision, recall and F1 of 84.56%, 84.66%, and 84.59%, respectively. XGB stood out with precision, recall, and F1 of 91.62%, 91.66%, and 91.62%. Among the three deep learning models compared, CNN outperformed LSTM and GRU in overall performance. This was possibly due to its stronger ability to capture local features and hierarchical information.

The proposed MoveRec significantly outperformed the nine different models compared, achieving precision, recall, and F1 of 95.66%, 95.50%, and 95.50%. MoveRec displayed a substantial improvement over the best-performing model XGB, which achieved precision, recall and F1 of 91.62%, 91.66%, and 91.62%. Additionally MoveRec improved precision by 4.04% (=95.66% - 91.62%), recall by 3.84% (=95.50% - 91.66%) and F1 by 3.88% (=95.50% - 91.62%) compared to XGB.

The results demonstrated the effectiveness of MoveRec in outperforming the traditional ML and deep learning models.

### *5.3.5. Result of RQ5*

To address RQ5, we compared MoveRec, with existing approaches for the move method. The experimental results are shown in Table 9. MoveRec demonstrated an increase of 28.4%(=78.5%-50.1%)-66.4%(=78.5%-14.1%) in precision and 9.4%(=74.0%-

**Table 9**
Comparison of MoveRec with existing refactoring methods (%).

| Project | PathMove | | | JDeodorant | | | JMove | | | RMove | | | MoveRec | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Precision | Recall | F1 | Precision | Recall | F1 | Precision | Recall | F1 | Precision | Recall | F1 | Precision | Recall | F1 |
| Weka | 22.4 | 64.5 | 33.2 | 5.9 | 54.8 | 10.7 | 10.8 | 74.1 | 18.9 | 50.8 | **95.9** | 66.4 | **91.4** | 91.0 | **91.0** |
| Ant | 19.7 | 56.0 | 29.2 | 17.1 | 48.0 | 25.2 | 17.3 | 84.0 | 28.7 | 49.1 | **90.5** | 63.7 | **80.1** | 78.4 | **79.2** |
| Maven | 54.5 | 54.1 | 54.3 | 13.9 | 25.0 | 17.9 | 10.4 | 37.5 | 16.3 | 49.6 | **85.3** | 62.7 | **86.8** | 84.8 | **84.4** |
| JTOpen | 41.6 | 51.2 | 45.9 | 20.7 | 44.7 | 28.3 | 20.8 | **89.4** | 33.7 | 48.4 | 87.2 | **62.3** | **60.6** | 58.8 | 57.1 |
| DrJava | 42.8 | 50.0 | 46.1 | 12.8 | 55.5 | 20.8 | 12.8 | 77.7 | 22.0 | 52.6 | **96.0** | **68.0** | **73.8** | 62.9 | 58.1 |
| Avg | 36.2 | 55.2 | 41.7 | 14.1 | 45.6 | 20.6 | 14.4 | 72.5 | 23.9 | 50.1 | **91.0** | 64.6 | **78.5** | 75.2 | **74.0** |

**Table 10**
Time of MoveRec for each component.

| Component | Time (m) |
|---|---|
| Generation of metric features | 5 |
| Generation of textual features | 126 |
| Generation of semantic features | 4 |
| Building the dataset | 1.5 |
| Building the classifier | 0.6 |
| Training | 119.6 |
| Testing | 0.4 |
| Total | 257.1 |

64.6%)-53.4%(=74.0%-20.6%) in F1 compared to existing refactoring tools: PathMove, JDeodorant, JMove and RMove. We highlight the best performance of each tool on 5 projects. Among them, JMove's Recall on JTOpen program was better than other tools, reaching 89.4%.

Compared to RMove, MoveRec had higher precision and F1 in 5 projects, but it was not as high performing as RMove in recall. A potential explanation for the analysis could be that the sample size utilized for the move method within the tested project was relatively limited, coupled with MoveRec's diverse feature extraction, which could collectively contribute to suboptimal prediction performance.

#### *5.3.6. Result of RQ6*

To answer RQ6, we evaluated the time cost of MoveRec during the refactoring of move method. For the selected dataset, we recorded the time of MoveRec's components as shown in Table 10. The total time is 257.1 minutes with 51.42 minutes per project on average. Most of the time is consumed by the training and generation of textual features, which accounts for over 95% of the total time. Generating textual features requires processing the data including converting textual information into feature vectors, which takes a considerable amount of time. The training involved multiple neural networks processing the features, making it complex and time-consuming. The total time to generate textual features is 126 minutes. Obtaining metric features through static analysis tools took an average of 1 minute per project while building the dataset and classifier took 1.5 minutes and 0.6 minutes, respectively.

### *5.4. Threats to validity*

The threats to validity primarily stem from the following aspects. The first threat comes from the dataset we evaluated. We only evaluated our method on a selected portion of open-source projects. To mitigate this issue, we choose 58 real Java development projects that cover multiple domains as comprehensively as possible. The second threat is that ChatGPT is a generative LLM, and the generated textual information is influenced by prompt design and parameter tuning. We addressed this by designing specific prompts to minimize the impact. The third threat was that the samples of move method refactoring in the projects we collect are relatively few. To reduce this threat, we used SMOTE to balance the dataset after cleaning the data, ensuring a balanced dataset.

## 6. Conclusion

This paper proposes MoveRec, a move method refactoring recommendation based on deep learning and LLM-generated information. This method uses an LLM as an auxiliary tool on top of traditional feature extraction, leveraging prompt learning to mine deep semantic information from the code and generate code summaries. By treating code summaries and changes in paths before and after refactoring as textual information, we calculate the similarity between the textual information changes and code summaries to uncover latent semantic relationships among different pieces of information. We then combined these insights with deep learning models to recommend refactoring on the move method. In our experiments, we compared the performance of traditional tools and LLM on textual information. We found that LLM is more effective than conventional tools in capturing the deep semantics behind the code and the true intentions of developers, with an average F1 improvement of 5.24%. Furthermore, we conducted comparative tests on existing move method refactoring methods in projects. The experimental results showed that MoveRec improves the average F1 ranging from 9.4% to 53.4%, effectively realizing refactoring recommendations on the move method. In future work, we will

recommend other refactoring methods to continue validating the applicability of our method and will also continue to improve our method to enhance recall during the refactoring recommendation process.

## CRediT authorship contribution statement

**Yang Zhang:** Writing – review & editing, Supervision, Methodology, Funding acquisition. **Yanlei Li:** Writing – original draft, Software. **Grant Meredith:** Writing – review & editing. **Kun Zheng:** Writing – review & editing. **Xiaobin Li:** 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.

## Acknowledgement

The authors also gratefully acknowledge the insightful comments and suggestions of the reviewers, which have improved the presentation. This work is partially supported by the Natural Science Foundation of Hebei Province under Grant No.F2023208001, and the Overseas High-level Talent Foundation of Hebei Province under Grant No.C20230358.

## Data availability

Data will be made available on request.

## References

[1] Apache, Jmeter, https://github.com/apache/jmeter, 2005.
[2] A. Barbez, F. Khomh, Y.-G. Guéhéneuc, A machine-learning based ensemble method for anti-patterns detection, J. Syst. Softw. 161 (2020) 110486.
[3] G. Bavota, et al., Methodbook: recommending move method refactorings via relational topic models, IEEE Trans. Softw. Eng. 40 (7) (2013) 671–694.
[4] T. Chen, C. Guestrin, Xgboost: a scalable tree boosting system, in: Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 2016, pp. 785–794.
[5] M. Cristina, M. Radu, F. Mihancea, iplasma: an integrated platform for quality assessment of object-oriented design, in: Proceedings of the 21st IEEE International Conference on Software Maintenance, IEEE, 2005, pp. 77–80.
[6] D. Cui, et al., Rmove: recommending move method refactoring opportunities using structural and semantic representations of code, in: 2022 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, 2022, pp. 281–292.
[7] Cui, Q.W. Jiaqi Wang, et al., Three heads are better than one: suggesting move method refactoring opportunities with inter-class code entity dependency enhanced hybrid hypergraph neural network, in: Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering (ASE), ACM, 2024, pp. 745–757.
[8] K. Depalma, et al., Exploring chatgpt's code refactoring capabilities: an empirical study, Expert Syst. Appl. 249 (2024) 123602.
[9] J. Devlin, M.-W. Chang, K. Lee, K. Toutanova, Bert: pre-training of deep bidirectional transformers for language understanding, in: Proceedings of NAACL-HLT, 2019, p. 2.
[10] R. Dey, F.M. Salem, Gate-variants of gated recurrent unit (gru) neural networks, in: 2017 IEEE 60th International Midwest Symposium on Circuits and Systems (MWSCAS), IEEE, 2017, pp. 1597–1600.
[11] M. Fowler, Refactoring: Improving the Design of Existing Code, Addison-Wesley, 1999.
[12] M. Hadj-Kacem, N. Bouassida, Deep representation learning for code smells detection using variational auto-encoder, in: 2019 International Joint Conference on Neural Networks (IJCNN), IEEE, 2019, pp. 1–8.
[13] T. Kattborn, et al., Review on convolutional neural networks (cnn) in vegetation remote sensing, ISPRS J. Photogramm. Remote Sens. 173 (2021) 24–49.
[14] M.U. Khattak, et al., Maple: multi-modal prompt learning, in: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2023, pp. 19113–19122.
[15] Z. Kurbatova, et al., Recommendation of move method refactoring using path-based representation of code, in: Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops, IEEE, 2020, pp. 315–322.
[16] Z.A. Lan, Albert: a lite bert for self-supervised learning of language representations, preprint, arXiv:1909.11942, 2019.
[17] M.P. Lavallee, Logistic regression, Circulation 117 (18) (2008) 2395–2399.
[18] J. Li, J.-h. Cheng, J.-y. Shi, F. Huang, Brief introduction of back propagation (bp) neural network algorithm and its improvement, in: Advances in CSIE, vol. 2, in: AISC, vol. 169, Springer-Verlag Berlin Heidelberg, 2012, pp. 553–558.
[19] B. Liu, et al., Deep learning based feature envy detection boosted by real-world examples, in: Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2023, pp. 908–920.
[20] H. Liu, et al., Deep learning based code smell detection, IEEE Trans. Softw. Eng. 47 (9) (2019) 1811–1837.
[21] Y. Liu, et al., Roberta: a robustly optimized bert pretraining approach, preprint, arXiv:1907.11692, 2019.
[22] Z. Liu, Y. Wang, S. Vaidya, F. Ruehle, J. Halverson, M. Soljačić, T.Y. Hou, M. Tegmark, Kan: Kolmogorov-Arnold networks, preprint, arXiv:2404.19756, 2024, Submitted on 30 Apr 2024 (v1), last revised 16 Jun 2024 (this version, v4).
[23] L. Ma, Y. Zhang, Using word2vec to process big text data, in: 2015 IEEE International Conference on Big Data (Big Data), IEEE, 2015, pp. 2895–2897.
[24] F. Palomba, et al., A textual-based technique for smell detection, in: 2016 IEEE 24th International Conference on Program Comprehension (ICPC), 2016, pp. 1–10.
[25] M. Pividori, Chatbots in science: what can chatgpt do for you?, Nature (2024), Epub ahead of print.
[26] S.J. Rigatti, Random forest, J. Insurance Med. 47 (1) (2017) 31–39.
[27] T. Sharma, Codesplitjava, https://github.com/tushartushar/CodeSplitJava, 2019.
[28] T. Sharma, et al., On the feasibility of transfer-learning code smells using deep learning, preprint, arXiv:1904.03031, 2019.
[29] Y.-Y. Song, Y. Lu, Decision tree methods: applications for classification and prediction, Shanghai Archiv. Psych. 27 (2) (2015) 130.

[30] D. Spadini, M. Aniche, A. Bacchelli, 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, 2018, pp. 908–911.
[31] S. Suthaharan, S. Suthaharan, Support vector machine, in: Machine Learning Models and Algorithms for Big Data Classification: Thinking with Examples for Effective Learning, Springer, 2016, pp. 207–235.
[32] R. Terra, et al., Jmove: a novel heuristic and tool to detect move method refactoring opportunities, J. Syst. Softw. 138 (2018) 19–36.
[33] N. Tsantalis, T. Chaikalis, A. Chatzigeorgiou, Jdeodorant: identification and removal of type-checking bad smells, in: 2008 12th European Conference on Software Maintenance and Reengineering (CSMR), IEEE, 2008, pp. 329–331.
[34] N. Tsantalis, A. Ketkar, D. Dig, Refactoringminer 2.0, IEEE Trans. Softw. Eng. 48 (3) (2020) 930–950.
[35] N. Ujihara, et al., c-jrefrec: change-based identification of move method refactoring opportunities, in: 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE, 2017, pp. 482–486.
[36] Y. Wang, H. Yao, S. Zhao, Y. Zheng, Dimensionality reduction strategy based on auto-encoder, in: Proceedings of the 7th International Conference on Internet Multimedia Computing and Service, 2015, pp. 1–4.
[37] E. Keogh, R. Miikkulainen, Naïve Bayes, Encyclopedia of Machine Learning, vol. 15, Springer, 2010, pp. 713–714.
[38] J. White, et al., Chatgpt prompt patterns for improving code quality, refactoring, requirements elicitation, and software design, in: Generative AI for Effective Software Development, Springer Nature, Switzerland, 2024, pp. 71–108.
[39] Z. Yang, Xlnet: generalized autoregressive pretraining for language understanding, preprint, arXiv:1906.08237, 2019.
[40] J. Yu, D. Tao, M. Wang, Y. Rui, Learning to rank using user clicks and visual features for image retrieval, IEEE Trans. Cybern. 45 (4) (2014) 767–779.
[41] Y. Yu, et al., A review of recurrent neural networks: Lstm cells and network architectures, Neural Comput. 31 (7) (2019) 1235–1270.
[42] P.L. Zhang, et al., Practices and challenges of using github copilot: an empirical study, preprint, arXiv:2303.08733, 2023.
[43] Y. Zhang, et al., Delesmell: code smell detection based on deep learning and latent semantic analysis, Knowl.-Based Syst. 255 (2022) 109737.