An Accurate Identifier Renaming Prediction and Suggestion Approach

JINGXUAN ZHANG, JUNPENG LUO, JIAHUI LIANG, LINA GONG, and ZHIQIU HUANG, Nanjing University of Aeronautics and Astronautics, China

Identifiers play an important role in helping developers analyze and comprehend source code. However, many identifiers exist that are inconsistent with the corresponding code conventions or semantic functions, leading to flawed identifiers. Hence, identifiers need to be renamed regularly. Even though researchers have proposed several approaches to identify identifiers that need renaming and further suggest correct identifiers for them, these approaches only focus on a single or a limited number of granularities of identifiers without universally considering all the granularities and suggest a series of sub-tokens for composing identifiers without completely generating new identifiers. In this article, we propose a novel identifier renaming prediction and suggestion approach. Specifically, given a set of training source code, we first extract all the identifiers in multiple granularities. Then, we design and extract five groups of features from identifiers to capture inherent properties of identifiers themselves and the relationships between identifiers and code conventions, as well as other related code entities, enclosing files, and change history. By parsing the change history of identifiers, we can figure out whether specific identifiers have been renamed or not. These identifier features and their renaming history are used to train a Random Forest classifier, which can be further used to predict whether a given new identifier needs to be renamed or not. Subsequently, for the identifiers that need renaming, we extract all the related code entities and their renaming change history. Based on the intuition that identifiers are co-evolved as their relevant code entities with similar patterns and renaming sequences, we could suggest and recommend a series of new identifiers for those identifiers. We conduct extensive experiments to validate our approach in both the Java projects and the Android projects. Experimental results demonstrate that our approach could identify identifiers that need renaming with an average F-measure of more than 89%, which outperforms the state-of-the-art approach by 8.30% in the Java projects and 21.38% in the Android projects. In addition, our approach achieves a Hit@10 of 48.58% and 40.97% in the Java and Android projects in suggesting correct identifiers and outperforms the state-of-the-art approach by 29.62% and 15.75%, respectively.

CCS Concepts: • **Software and its engineering → Software libraries and repositories**; **Software post-development issues;**

Additional Key Words and Phrases: Identifier renaming, source code analysis, code refactoring, mining code repository

This work was supported in part by the National Natural Science Foundation of China under grant 62272225 and the Fund of Prospective Layout of Scientific Research for NUAA (Nanjing University of Aeronautics and Astronautics).
Authors' address: J. Zhang, J. Luo, J. Liang, L. Gong, and Z. Huang, Nanjing University of Aeronautics and Astronautics, Yudao Street 29th, Nanjing, Jiangsu, China, 210016; emails: {jxzhang, luojunpeng, liangjiahui, gonglina, zqhuang}@nuaa.edu.cn.

1049-331X/2023/09-ART148 $15.00
https://doi.org/10.1145/3603109

ACM Reference format:
Jingxuan Zhang, Junpeng Luo, Jiahui Liang, Lina Gong, and Zhiqiu Huang. 2023. An Accurate Identifier Renaming Prediction and Suggestion Approach. *ACM Trans. Softw. Eng. Methodol.* 32, 6, Article 148 (September 2023), 51 pages.
https://doi.org/10.1145/3603109

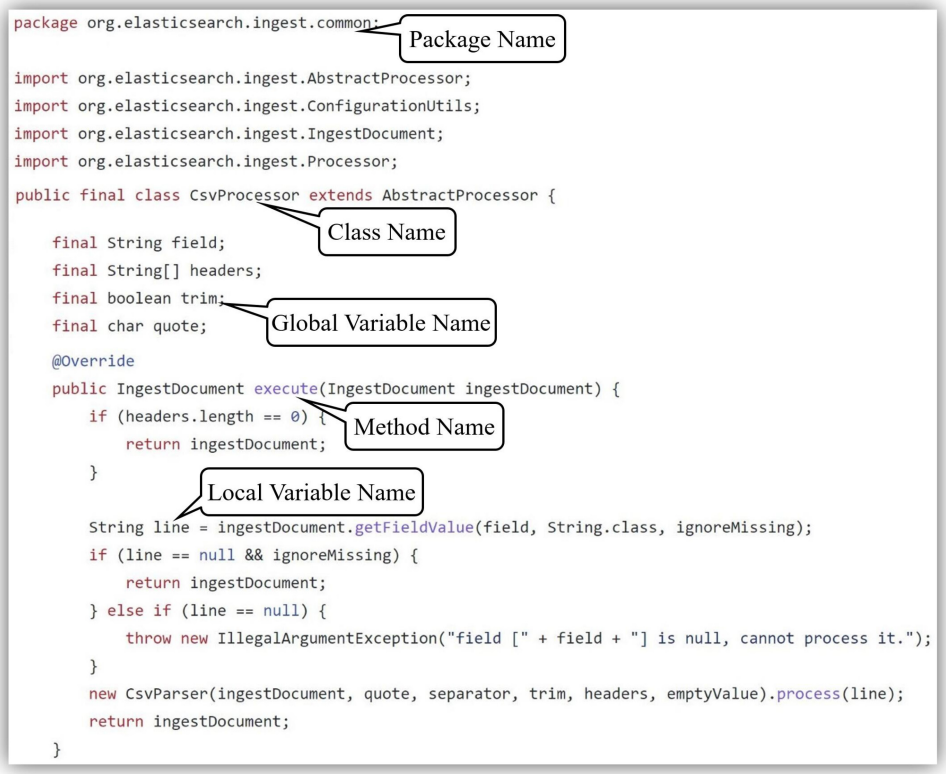
1 INTRODUCTION

Source code analysis and comprehension is a common activity conducted by developers in their daily development process [38]. To guarantee the efficiency of source code analysis, developers usually rely on source code lexicon, especially identifiers [60]. Hence, identifiers are critical for developers. However, existing studies have shown that identifiers are not always of high quality [17, 27]. On the one hand, due to the inexperience, carelessness, and heavy workload of novice software engineers, they may define a lot of flawed identifiers. Flawed identifiers are those identifiers that violate their naming guidelines, semantic roles, and programming context so that they need to be renamed [31]. On the other hand, along with the evolution of source code, the literal meanings of some obsolete identifiers may no longer match their semantic functions [4]. Hence, plentiful flawed identifiers exit in source code, which could mislead developers from correctly comprehending source code, leading to potential software defaults [1].

The flawed identifiers have already caused serious software development problems in practice (see Section 2.1) [10, 13, 54]. Hence, to ensure the quality of identifiers, developers need to rename them regularly [2, 12]. Researchers have conducted a survey to investigate developers' practical renaming habit in real software development [7]. From the perspective of renaming frequency, they found that 39% of developers conducted renaming from a few times a week to almost every day. In addition, 46% of developers performed renaming a few times a month. In terms of renaming tool support, researchers found that 20% of developers renamed code entities manually. Even though 72% of developers employed tools to rename code entities, 92% of developers thought that renaming was not straightforward. As for the renaming costs, 67% of developers regarded that renaming required considerable time and effort and also depended on particular cases. Hence, manually identifying the exact identifiers that need renaming from a large amount of identifiers is tedious and time consuming for developers, especially in the context of cross-regional and cross-project software development [19, 38].

There are two reasons making identifier renaming challenging [7, 34]. First, the identifiers needing renaming only take a tiny fraction in all the identifiers in source code. According to our statistics on the experimental projects in this study, the overall percentage of identifiers that need renaming take up 6.36% and 7.27% in the Java projects and the Android projects, respectively. It should be noted that a small percentage does not mean that identifiers needing renaming are not important. On the one hand, considering that identifiers take up almost 70% of source code lexicon, the number of identifiers that need renaming is still large. On the other hand, these identifiers that need renaming could cause development problems and identifying these identifiers depends on developers' development experience. Hence, picking up these identifiers is tough. Second, in Figure 1, a real piece of code from a popular project, *ElasticSearch*,[1] is shown, whose function is to break lines from CSV files into separate fields. We can see that there are different types of identifiers in this example. Package names, type (class and interface) names, method names, global variable (field) names, and local variable names are all identifiers. In this article, we employ the

[1]https://github.com/elastic/elasticsearch/blob/main/modules/ingest-common/src/main/java/org/elasticsearch/ingest/common/CsvProcessor.java.




```
package org.elasticsearch.ingest.common;

import org.elasticsearch.ingest.AbstractProcessor;
import org.elasticsearch.ingest.ConfigurationUtils;
import org.elasticsearch.ingest.IngestDocument;
import org.elasticsearch.ingest.Processor;

public final class CsvProcessor extends AbstractProcessor {

    final String field;
    final String[] headers;
    final boolean trim;
    final char quote;

    @Override
    public IngestDocument execute(IngestDocument ingestDocument) {
        if (headers.length == 0) {
            return ingestDocument;
        }
        String line = ingestDocument.getFieldValue(field, String.class, ignoreMissing);
        if (line == null && ignoreMissing) {
            return ingestDocument;
        } else if (line == null) {
            throw new IllegalArgumentException("field [" + field + "] is null, cannot process it.");
        }
        new CsvParser(ingestDocument, quote, separator, trim, headers, emptyValue).process(line);
        return ingestDocument;
    }
```

Fig. 1. A real example of different granularities of identifiers.

term *granularity* to stand for these different types of identifiers (e.g., local variable name). Different granularities of identifiers have various morphologies [53]. Hence, it is hard to universally propose a method to identify all the granularities of identifiers that need renaming. Furthermore, automatically generating and recommending a new full name identifier to replace the old flawed identifier is also a difficult research task, since the new identifier should not only be composed of correct constitutive terms with correct **Part of Speech (POS)** but also should follow the corresponding identifier styles with suitable length [56]. Hence, automatic identifier renaming prediction and suggestion techniques urgently need to be proposed.

To assist identifier renaming, researchers have proposed several approaches to predict those identifiers that need renaming and suggest new identifiers for them [39, 41]. However, these approaches usually only focus on a specific granularity of identifiers. In addition, these existing approaches can only suggest sub-tokens of identifiers without considering their orders in identifiers, which may be of limited help to developers. In such a suggestion, developers shall decide the sequence of suggested sub-tokens and the corresponding identifier style, which may also be difficult for developers to choose [21, 33]. Recently, two identifier renaming prediction approaches are proposed and achieve promising results, namely *DeepMethod* [36] and *RenameExpander* [39]. *DeepMethod* is a typical deep learning based approach. It embeds identifiers into vectors, employs textCNN to learn a classifier, and leverages historical renaming to refine the prediction results of the trained classifier. *RenameExpander* leverages the conducted historical renaming to decide the future renaming of identifiers, with the rationale that once an identifier is renamed,

similar identifiers shall also be renamed. In addition, a new approach called *RefactorLearning* [41] is proposed for suggesting correct new identifiers for those that need renaming. It is also a deep learning based approach, whose basic idea is to search and recommend similar identifiers in the deep semantic vector space. However, these existing approaches do not fully leverage the domain-specific information (especially they do not systemically analyze why and how developers rename identifiers), so there is room for improvement.

To tackle the preceding shortcomings of existing approaches and better predict flawed identifiers as well as suggest accurate full name identifiers, we propose a novel approach leveraging semantic relationships between identifiers and related code entities as well as the historical renaming activities. Specifically, our approach consists of two phases: the prediction phase and the suggestion phase. In the prediction phase, given a set of training projects, we first extract all the granularities of identifiers and their corresponding change history. If an identifier has been renamed in history, we trace it back to its historical version so that we could predict whether it needs renaming and suggest new identifiers for it based on the correct context. Then, we define and extract five groups of features measuring the lexical characteristics and semantic relationships between the current identifier and relevant code entities as well as code conventions. These features could detect whether the current identifier has semantic conflict with related code entities and conventions. These features of identifiers and corresponding labels (whether they need renaming or not) are used to train a Random Forest classifier. Finally, after the classifier is fully trained, it can be used to predict whether a specific identifier needs renaming or not. In the suggestion phase, for the identifier needing renaming, we first find all the related code entities and their change history. Based on the intuition that identifiers are co-evolved with their relevant code entities, we suggest and recommend a series of new identifiers for this given identifier based the historical and current version of related code entities.

We collect a dataset containing 100 open source projects from GitHub, including 50 Java projects and 50 Android projects. Based on this dataset, we conduct extensive experiments to validate the effectiveness of our proposed approach. From the experimental results, we find that our approach can predict the exact identifiers that need renaming with an average F-measure of more than 89% in both the Java and Android projects, which outperforms the state-of-the-art approach by up to 21.38%. Along with the increase of the number of features, our approach achieves an overall upward trend in terms of Precision, Recall, and F-measure. Even in the context of cross-project prediction, our approach achieves the F-measure of more than 80%. In addition, our approach achieves a Hit Ratio of 48.58% and 40.97% when recommending 10 full name identifiers in the Java and Android projects, respectively, which also outperforms the state-of-the-art approach by 29.62% and 15.75%.

The main contributions of this study are summarized as follows:

(1) We propose a novel identifier renaming prediction and suggestion approach leveraging semantic features and related code change history.
(2) To the best of our knowledge, this is the first approach targeted toward multiple granularities of identifiers. We open the source code of our approach to facilitate replication and future extension.[2]
(3) We conduct extensive experiments to show the effectiveness of our approach. Experimental results demonstrate that our approach is superior to the existing state-of-the-art approaches.

The rest of the article is structured as follows. In Section 2, we explain the motivation for this work. In Section 3, we detail our proposed approach with its main components. In Sections 4 and

[2]https://github.com/APIDocEnrich/Renaming.

Table 1. Search Results in Stack Overflow and GitHub

Search Query	Stack Overflow	GitHub				
	Question	Repository	Open Issues	Closed Issues	All Issues	Discussion
Incorrect (not correct) Identifier	20 (0.13%)	242	65,362	371,370	436,732	1,704
Inconsistent (not consistent) Identifier	2 (0.01%)	67	24,360	113,707	138,067	470
Unexpected (not expected) Identifier	766 (4.95%)	183	58,229	183,940	242,169	1,144
Invalid (not valid) Identifier	841 (5.43%)	172	70,560	324,867	395,427	1,283
Failed Identifier	62 (0.40%)	146	26,945	173,171	200,116	818
Undeclared Identifier	848 (5.48%)	5	10,805	42,188	52,993	142
Wrong Identifier	48 (0.31%)	80	34,833	141,996	176,829	1,064
Bug Identifier	8 (0.05%)	225	58,113	441,211	499,324	839
Error Identifier	2,185 (14.12%)	592	152,861	631,531	784,392	2,671
Sum	4,780 (30.88%)	1,712	502,068	2,423,981	2,926,049	10,135

Verified on August 8, 2022.

5, we show the experimental setup and experimental results, respectively. We then illustrate the threat to validity and related work in Sections 6 and 7. Finally, we make a summary and point out future work of this study in Section 8.

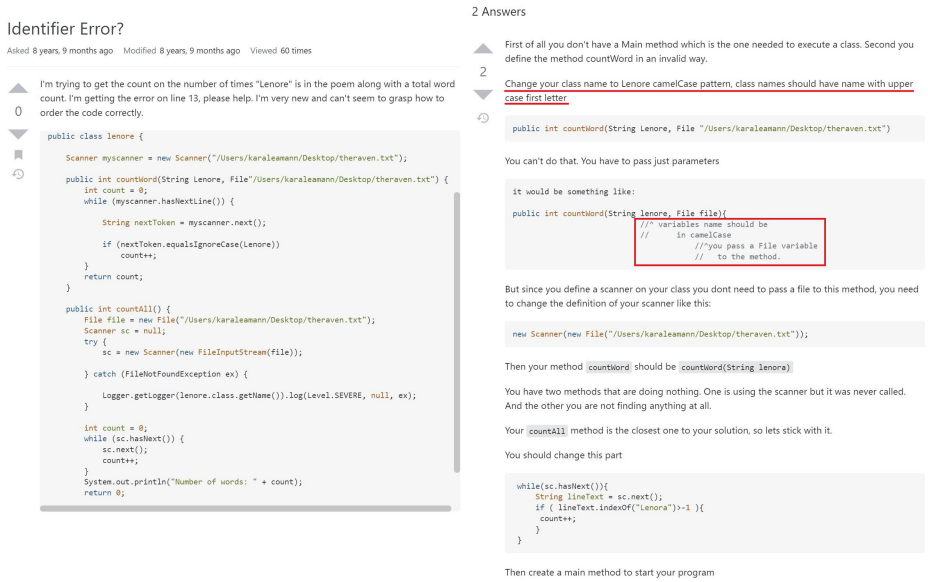
2 MOTIVATION

In this section, we present the motivation for this work. First, we explore the serious problems caused by the flawed identifiers. Then, we investigate why developers rename identifiers, based on which we propose our identifier renaming prediction algorithm. Finally, we examine how developers rename flawed identifiers, based on which we propose our identifier renaming suggestion algorithm.

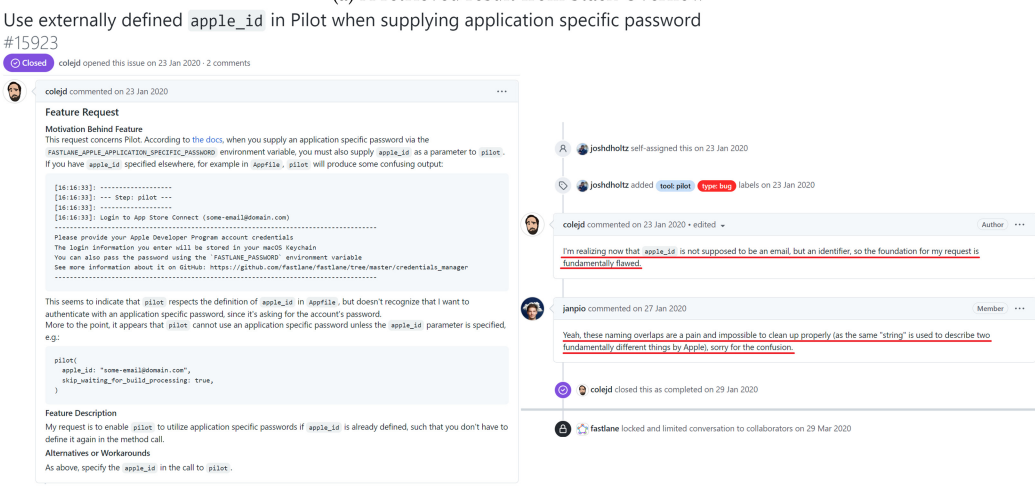
2.1 Impacts of Flawed Identifiers

Flawed identifiers usually cause serious software development problems, including triggering software bugs and increasing software maintenance costs [38, 59]. To explore such broad impacts, we conduct an empirical study. Specifically, we first define some keywords related to flawed identifiers as search queries. We combine the word "identifier" with words describing the flawed aspects, including "incorrect (not correct)," "inconsistent (not consistent)," "unexpected (not expected)," "invalid (not valid)," "failed," "undeclared," "wrong," "bug," and "error." In such a way, we generate a series of search queries, such as "failed identifier" and "error identifier." Then, we employ these queries to search relevant questions in Stack Overflow and related development activities in GitHub. Both Stack Overflow and GitHub provide their own search engines, which make it easy for us to perform the search and analysis. When we search Stack Overflow, we only search these queries within the titles of questions. This is because the title is a good summary of the whole question. Once a search query appears in the title of a question, we regard that this question focuses on the discussion of the impacts as well as solutions of flawed identifiers. This strategy could help us filter out irrelevant questions and obtain more exact questions. In addition, since there are plentiful questions in Stack Overflow, counting only the number is meaningless. We also obtain the questions related to the identifier itself by searching the keyword "identifier" in the question title and calculate the percentages of questions related to flawed identifiers within all the questions related to identifiers in Stack Overflow. When we search GitHub, we count the number of repositories, open and closed issues, and discussions. By analyzing the results of Stack Overflow and GitHub, we can evaluate the impacts of flawed identifiers to software development.

Table 1 presents our results in Stack Overflow and GitHub for each defined query. We can see that 4,780 questions related to flawed identifiers can be retrieved in Stack Overflow. These questions take up 30.88% of those related to identifiers. Among them, nearly half (i.e., 2,185) of the



(a) A retrieved result from Stack Overflow



(b) A retrieved result from GitHub

Fig. 2. The retrieved results in Stack Overflow and GitHub.

questions are related to the search query "error identifier." This means that developers often discuss flawed identifier related questions in Stack Overflow. In terms of the development activities in GitHub, we can obtain 1,712 repositories, more than 2,926,000 issues, and 10,135 discussions in total with these search queries. Among the retrieved issues, 502,068 issues are still open and the others are closed. This shows that flawed identifier related issues are continuously emerging in the process of software development. From Table 1, we can see that flawed identifiers have a broad impact on software developers and software development process.

To better present such an impact, we provide two real examples obtained from Stack Overflow and GitHub, respectively, as shown in Figure 2. Figure 2(a) is the example from Stack Overflow.[3]

[3]https://stackoverflow.com/questions/20485815/identifier-error.

A new developer who does not have a lot of experience in programming asks a question to verify whether the defined identifiers are correct or not by providing a piece of code. The aim of the new developer is to count the frequency of occurrence of a specific term *Lenore*, as well as the total number of words in a long text. However, he or she has some difficulties to implement this functionality, leading to a bug in the piece of code. He or she guesses that the defined identifiers may be incorrect, so he or she asks this question. After that, an experienced developer tries to help the submitter of the question fix the bug. As the submitter of the answer says, there are several irregular identifiers that do not follow the corresponding styles or cases, which need to be renamed. In addition, it is also incorrect to define real values as the parameters—for example, the file "theraven.txt" should not be passed as the parameters. From this example, we can see that new developers often define irregular identifiers, leading to the possible bugs and misunderstanding of the code. Figure 2(b) shows a real bug or feature request in GitHub,[4] which actually should not be submitted due to the misunderstanding of the submitter to the functionalities of identifiers. As the submitter points out, when developers supply an application specific password via the environment variable, they must also supply *apple_id* as a parameter to *pilot*. Otherwise, the program will produce some confusing outputs. Hence, the submitter proposes a request to enable *pilot* to utilize application specific passwords, if *apple_id* is already defined. However, after that, he or she realizes that *apple_id* is not supposed to be an email but an identifier, so the foundation for the request is flawed. Another developer also confirms that this easily leads to confusion since there are meaning overlaps, which are hard to clean up properly. This example demonstrates that inappropriate identifiers often confuse developers, leading to the potential software bugs.

2.2 Why Developers Rename Identifiers

In the literature, different researchers have reported different reasons developers rename identifiers [7, 47, 49]. We perform a literature analysis on the related studies about identifier renaming and summarize these reasons, whose results are shown in Figure 3. Specifically, we first employ the phrase "identifier rename reasons" as the query, as it is directly related to the reasons of identifier renaming. Then, we input it to Google Scholar to obtain relevant scientific papers. Next, we download and manually check the top 50 ranked papers returned by Google Scholar. We regard that these papers are highly related to the query, from which we can extract the reasons for identifier renaming. For the top 50 ranked papers, the first three authors of this paper read the content of these papers independently to discover whether the reasons for identifier renaming are mentioned explicitly. If we find any content that explains the reasons for identifier renaming, we record the reasons. Finally, through an in-person discussion, the three authors summarize and merge the results to obtain the final reasons for identifier renaming. It should be noted that when we generate the reasons of identifier renaming, we only retain those reasons that are general and can be applied to all granularities of identifiers. If a specific reason is only targeted toward a specific granularity, this reason is removed and not considered in this study.

As a result, we collect 12 papers that explicitly point out some reasons for identifier renaming. Based on these papers, we divide the reasons for identifier renaming into two dimensions, including lexical reasons and semantic reasons. For the lexical reasons, we find that developers rename identifiers to adhere to code conventions, fix typos, reformat, and both apply and expand abbreviations. First, some identifiers do not follow the corresponding code conventions proposed by the programming language or the specific projects, so these identifiers are renamed to make them consistent with code conventions [2]. For example, Lin et al. [37] investigated the quality of identifiers in test code. They indicated that a lot of identifiers do not follow code conventions in practice.

[4] https://github.com/fastlane/fastlane/issues/15923.

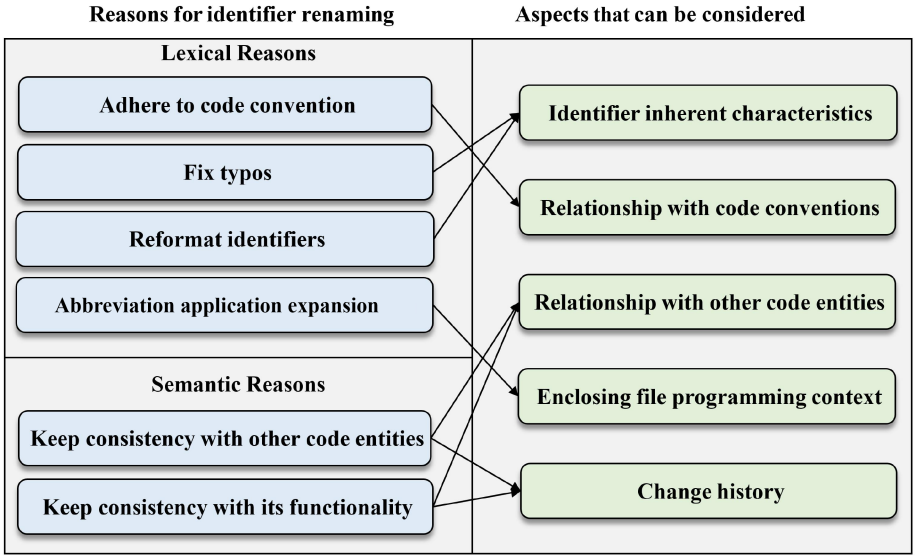


Fig. 3. Reasons for identifier renaming and the corresponding aspects can be considered for renaming prediction.

Hence, it is necessary to rename identifiers into more consistent ones. Second, some identifiers are initially wrongly named, so these identifiers also need renaming [20]. For example, as discussed by Arnaoudova et al. [7], spelling errors in identifiers mislead developers when they try to understand the rationale of identifiers. Hence, correcting spelling errors is also a reason for identifier renaming. Third, developers rename identifiers because they want to reformat them—for example, change identifier styles (camel case and snake case) or reorder constitutive terms of identifiers. For example, both Peruma et al. [48] and Arnaoudova et al. [7] mentioned that reformatting identifiers is an important lexical reason for renaming. In their studies, reformatting identifiers and reordering terms are two separate reasons. However, in this study, we merge the two reasons together, since we believe that they are closely related (term reordering is a types of reformatting). Finally, since some identifiers contain abbreviations or some identifiers are too long to be shortened, developers rename identifiers to make transitions between abbreviations and their expansions for composing terms. For instance, Pantiuchina et al. [47] conducted a large-scale study investigating why developers perform refactoring (including identifier renaming) in open source projects. They found that expanding abbreviations and shortening identifiers are another important reasons for identifier renaming.

Basically, for the semantic reasons, we discover that developers rename identifiers because these identifiers are in conflict with other code entities and their corresponding functionality [7, 49]. First, the definition of some identifiers are not consistent with other code entities, or once other related code entities are renamed, these identifiers shall be also renamed [31]. For example, Liu et al. [39] pointed out that once a renaming is conducted, the closely related identifiers have a large probability of being renamed. The rationale is that if a developer makes a mistake in naming an identifier, it is likely for him or her to make the same mistake in naming closely related identifiers. Second, along with the evolution of software, the definition of some identifiers may not well represent their semantic functionality or responsibility [9, 45]. In such a situation, these identifiers also need renaming. For example, Li et al. [34] pointed out that developers sometimes compose identifiers casually due to a constrained schedule. In such a situation, casual identifiers

are usually not representative of their role and not well thought out. In addition, software evolution also makes meaningful identifiers obsolete, again making the original identifiers not match the functionality.

From these reasons of identifier renaming, we can gain some aspects that could influence identifier renaming, which could also help us design an identifier renaming prediction algorithm. As shown in Figure 3, we can at least obtain five aspects that shall be considered from these identifier renaming reasons—that is, identifier inherent characteristics, relationship with code conventions, relationship with other code entities, enclosing file capturing programming context, and change history. The five aspects can be triggered or linked to the identified identifier renaming reasons. For example, for fixing typos and reformatting, we shall fully analyze the inherent characteristics of identifiers themselves when we design our identifier renaming prediction approach. Especially, we need to parse the inherent characteristics of identifiers, such as the styles and the composing terms, to find whether there are typos so as to better detect those identifiers that need renaming. For adherence to code convention, we need to check the relationships between identifiers and code conventions to detect whether there are conflicts so that they need to be renamed. In addition, if we want to perform the abbreviation application and expansion for identifiers, we shall detect the programming context to find the exactly correct expansions in the enclosing files. Hence, we shall also analyze the programming context of identifiers in the enclosing files. For keeping consistency with other code entities and their functionality, we shall consider the relationship between identifiers and their related code entities (e.g., whether there exist identifiers with the same name) as well as their corresponding change history (e.g., whether there are closely related identifiers that have been renamed in history) when we design our approach [14, 57]. Hence, by fully considering the five aspects (which can be easily transformed into features) of identifiers, we may improve the performance of identifier renaming prediction.

2.3 How Developers Rename Identifiers

After exploring the impacts of flawed identifiers and reasons for identifier renaming, we continue to explore how developers rename identifiers by checking the renaming activities conducted by developers in practice, which could also provide some hints for the better design of our identifier renaming suggestion approach. Specifically, we randomly sample 500 commits containing the keyword "rename identifier" in GitHub, and two authors of this article manually and independently check the corresponding commit messages and code changes. After both of the two authors finish the manual checking, they discuss the manual results and obtain consistent results. By inspecting these commits, it is our hope to find some common changes made to identifiers when developers rename identifiers [3].

As a result, for lexical identifier renaming, we find that developers tend to change styles of identifiers and expand abbreviations of composing terms [44]. For semantic identifier renaming, we discover that developers narrow, broaden, preserve, add, and remove semantic meanings of identifiers. When performing both lexical and semantic renaming for identifiers, developers unintendedly follow some renaming patterns based on a series of renaming sequences.

To better present the renaming patterns and renaming sequences, we find two real renaming examples, which are shown in Figure 4. Figure 4(a) shows the code change of a *switch* statement, from which we can find some renaming patterns. For this code change, developers change the names of three invoked methods, with one renaming change from *CreateCatmullRomPath* to *createCatmullRomPath* and two renaming changes from *CreateLinearPath* to *createLinearPath*. Obviously, these renaming changes follow some patterns, namely changing the first letter to lowercase or changing their styles from the upper camel case to the lower camel case. When analyzing a large scale of

```
132            switch (spline.getType()) {                    132            switch (spline.getType()) {
133                case CatmullRom:                           133                case CatmullRom:
134 -                  debugNode.attachChild(CreateCatmullRomPath());  134 +          debugNode.attachChild(createCatmullRomPath());
135                    break;                                 135                    break;
136                case Linear:                               136                case Linear:
137 -                  debugNode.attachChild(CreateLinearPath());  137 +          debugNode.attachChild(createLinearPath());
138                    break;                                 138                    break;
139                default:                                   139                default:
140 -                  debugNode.attachChild(CreateLinearPath());  140 +          debugNode.attachChild(createLinearPath());
141                    break;                                 141                    break;
142            }                                              142            }
```

(a) An example of renaming pattern

```
11  -   private String name;                                11  +   private String organizationName;
12      private String organizationKey;                     12      private String organizationKey;
13                                                          13
14      public KeyOrganization() {                          14      public KeyOrganization() {
15      }                                                   15      }
16                                                          16
17  -   @Column(name = "name", nullable = false, unique = true)  17  +   @Column(name = "organization_name", nullable = false, unique = true)
18  -   public String getName() {                           18  +   public String getOrganizationName() {
19  -       return name;                                    19  +       return organizationName;
20      }                                                   20      }
21                                                          21
22  -   public void setName(String name) {                  22  +   public void setOrganizationName(String organizationName) {
23  -       this.name = name;                               23  +       this.organizationName = organizationName;
24      }                                                   24      }
```

(b) An example of renaming sequence

Fig. 4. Two examples of how developers rename identifiers.

renaming changes, we may summarize some renaming patterns. These change patterns are useful for us to propose a new approach to better suggest new identifiers.

Figure 4(b) shows the code change of a defined class, from which we can see that renaming changes have their own sequences. First, the global variable (field name) is changed from *name* to *organizationName* to make it more specific and concrete. After changing the global variable name, its corresponding *setter* method (including the formal parameter) and *getter* method should also be renamed. We can see from this example that renaming changes have their sequences. By mining the change history of a project, we may generate a series of renaming sequences, which could also help us better design the new identifier suggestion approach.

In summary, by taking both the renaming patterns and renaming sequences (history) of related code entities, we may generate and recommend the correct full name identifiers for those which need renaming.

3 FRAMEWORK

In this section, we present the whole framework of our proposed approach with its main components in details.

Figure 5 shows the overall framework of our proposed approach. As shown in the figure, this novel approach consists of two phases: renaming prediction and renaming suggestion. The two phases are consecutive, and they are combined to form an overall framework for the prediction and suggestion of identifiers that need renaming.

The aim of the renaming prediction phase is to identify the exact identifiers that need renaming in all the granularities. The renaming prediction follows the typical supervised learning process, which means that it includes the training and prediction [51, 62]. In the training, given a source code repository, we first extract all the granularities of identifiers with their change history. If a specific identifier has been renamed in history, we trace back to its historical version so that we

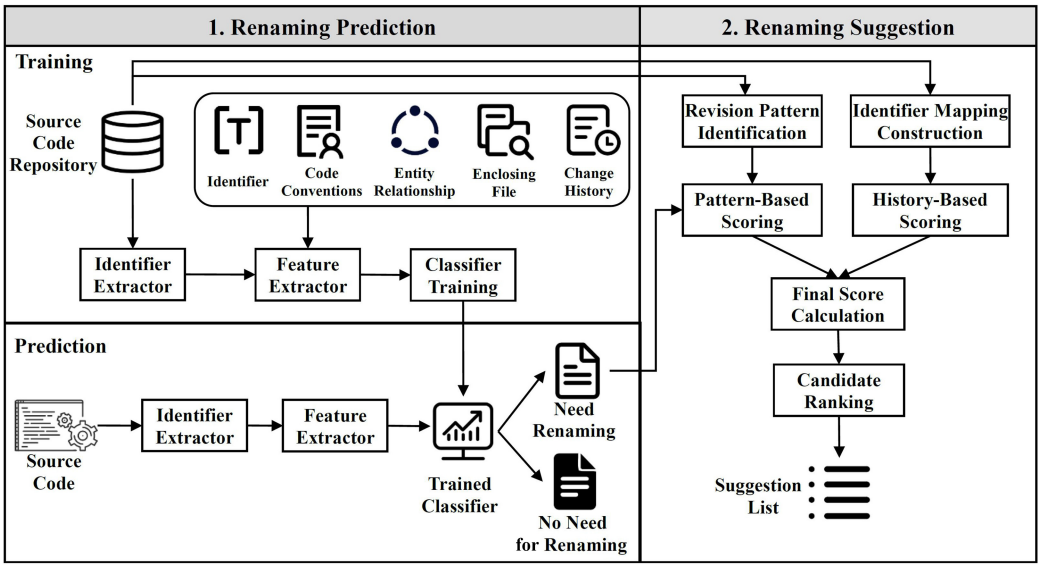


Fig. 5. The overall framework of our proposed approach.

could predict renaming based on the correct programming context and validate the correctness of the suggested new identifiers. Then, we define and extract five groups of features to measure the properties of identifiers themselves and the semantic relationships between identifiers and relevant code entities, as well as code conventions. These features could detect the definition conflict of identifiers so that they can help pick up those identifiers that need renaming from all the identifiers [19]. These identifier features and their corresponding labels (i.e., whether they need to be renamed or not) are used to fully train a classifier. Given a new piece of source code, after extracting identifiers and their corresponding features, this trained classifier can be used to predict whether these identifiers need to be renamed.

The aim of the renaming suggestion phase is to recommend a new identifier suggestion list for those identifiers in need of renaming that are identified in the first phase. In the renaming suggestion phase, we generate a series of new identifiers from two aspects. In the first aspect, we find that developers rename identifiers following specific patterns in Section 2. Hence, we recognize a set of revision patterns for constitutive tokens of identifiers. Based on these revision patterns, we can change the current identifier that needs renaming and generate a series of new identifiers with their relevant scores. In the second aspect, we also find that developers rename identifiers following a series of renaming sequences in Section 2. Hence, we obtain relevant code entities and their change history. Based on the relevant code entity change history, we can also generate a set of new identifiers with their relevant scores for the current identifier needing renaming. We combine the two scores and calculate the final relevant score for each of the generated new identifiers. Finally, based on the final score, we rank the newly generated identifiers to form a suggestion list. In the following section, we present the details of main components in the workflow of our approach in Figure 5.

3.1 The Renaming Prediction Phase

As shown in the overall framework, the renaming prediction phase mainly contains three components: the Identifier Extractor, the Feature Extractor, and Classifier Training. In this section, we present the details of these components.

3.1.1 Identifier Extractor. Given a piece of source code, we first transform it into an **Abstract Syntax Tree (AST)**. By traversing all the nodes in the AST, we can obtain all the contained identifiers with their corresponding granularities. In this study, we take all the granularities of identifiers into consideration—that is, package names, type names, method names, global variable names, and local variable names. Among them, type names include class names, interface names, enumeration names, and annotation names. In addition, formal parameters defined with the declaration of methods are regarded as local variables, since they can only be used within their corresponding methods. By parsing the node types of the generated AST, we can obtain the exact granularity for each identifier. This component is implemented by employing an existing widely used source code parser, namely the Java Parser.[5] This tool could help us easily transform source code into the AST and further extract all the granularities of identifiers.

3.1.2 Feature Extractor. As we described above, identifiers at different granularities may evolve differently. Hence, we need to measure and capture multiple aspects of identifiers. Motivated by the reasons of identifier renaming and the corresponding aspects that can be considered for renaming prediction in Section 2.2, we define and extract five groups of features of identifiers. Before we extract these features, we need to divide identifiers into a set of tokens for the ease of extracting the exact values of features. As the pre-order operation of feature extraction, dividing identifiers could help to easily analyze the compositions and lexical meanings of identifiers. In this study, we employ the **Identifier Name Tokenizer Tool**[6] (INTT) for splitting identifiers into their component tokens. INTT is a popular tokenizer that tokenizes identifiers using conventional typographical boundaries (e.g., camel case) and unconventional boundaries (e.g., the uppercase to lowercase transition). It performs well even when identifiers contain digits and single case strings with terms and abbreviations.

These proposed features capture both the lexical and semantic of the identifier themselves (identifier change history) and the relationships between identifiers and other software artifacts (e.g., related code entities). Based on the relationships of these features, we divide them into five groups, including the inherence group, the relation with convention group, the relation with code entities group, the enclosing file group, and the history group. Some of these features are from our literal review by examining the proposed features in related studies [11, 58]. The other features are from our observation of the renaming patterns of identifiers by manually checking the development activities of developers. Table 2 summarizes all five groups of features used for identifier renaming prediction. We detail the exact meaning, the rationality, and the calculation procedure for each feature as follows.

Inherence. The Inherence feature group measures the main characteristics of identifiers themselves. These main characteristics of identifiers could expose some clues to help us predict whether they need renaming or not [28]. This feature group includes 10 features in total, measuring the lengths, compositions, and styles of identifiers.

Granularity. This feature checks the exact granularity of the current identifier. Different granularities of identifiers may show different evolution patterns and renaming modes, so it is necessary to distinguish the granularity of identifiers. In this study, the granularity of identifiers is classified into five types, including package names, type names, method names, global variable names, and local variable names. Hence, the data type of the value of this feature is the categorical value. By parsing the node type of the AST, we can obtain the exact granularity of a specific identifier.

Length. This feature checks the number of characters a specific identifier contains. The lengths of identifiers should be in a reasonable range. Too long identifiers can influence the coding efficiency,

[5] https://javaparser.org/.
[6] https://github.com/sjbutler/intt.

Table 2. Summary of Defined Features Used by Identifier Renaming Prediction

Group	Feature	Description	Data Type
Inherence	Granularity	The exact granularity the identifier belongs to (e.g., global variable names)	Categorical value
	Length	The length of the identifier at character level	Numerical value
	Size	The number of constitutive tokens the identifier contains	Numerical value
	ContainsVerb	Whether the identifier contains verb tokens	Binary value
	ContainsNoun	Whether the identifier contains noun tokens	Binary value
	ContainsAdj	Whether the identifier contains adjective tokens	Binary value
	Style	The style of the identifier (e.g., camel case)	Categorical value
	Contains$	Whether the identifier contains the special character $	Binary value
	StartWithUpper	Whether the identifier starts with an uppercase letter	Binary value
	DigitCount	The number of digits the identifier contains	Numerical value
Relation with Convention	FollowConPOS	Whether the identifier follows the POS defined in code convention	Binary value
	FollowConStyle	Whether the identifier follows the Style defined in code convention	Binary value
	FollowCommonPOS	Whether the identifier follows the common POS	Binary value
	FollowCommonStyle	Whether the identifier follows the common style	Binary value
Relation with Code Entities	ContainOthers	Whether the identifier contains other identifiers	Binary value
	ContainedByOthers	Whether the identifier is contained in other identifiers	Binary value
	SameName	Whether there are identifiers with the same name and granularity	Binary value
	SameNameDiffGra	Whether there are identifiers with the same name but different granularity	Binary value
	ContainFather	Whether the identifier contains the name of its father	Binary value
	MinEdit	The minimal edit distance between this identifier and other identifiers	Numerical value
	MaxJaccard	The maximal Jaccard similarity of this identifier and other identifiers	Numerical value
Enclosing File	LOC	The LOC of the enclosing file of the identifier	Numerical value
	ImportCount	The import count of the enclosing file of the identifier	Numerical value
	CommentCount	The comment count of the enclosing file of the identifier	Numerical value
	ConditionCount	The conditional statement count of the enclosing file of the identifier	Numerical value
	IterationCount	The iteration statement count of the enclosing file of the identifier	Numerical value
History	ChangeCount	The change count of the identifier in history	Numerical value
	StateChangeCount	The change count of the statement enclosing the identifier in history	Numerical value
	FileChangeCount	The change count of the enclosing file of the identifier	Numerical value

whereas too short identifiers are difficult to understand. We propose this feature because identifiers that need renaming and identifiers that do not need renaming may have different lengths at the character level. This feature is calculated by counting how many characters there are in this identifier.

Size. This feature measures the number of constitutive tokens a specific identifier contains. Similarly, the number of constitutive tokens should also be in a feasible range. If the specific identifier is too long or too short, it may have a higher probability of being renamed. This feature can be calculated by summing up the number of tokens in the identifier, after it is split by INTT.

The feature *Length* and the feature *Size* are different. Specifically, the feature *Length* measures the scale at the character level, whereas the feature *Size* measures the scale at the token level for the identifier. For example, given the identifier *getConnectionResult*, the value of the feature *Length* is 19 since there are 19 characters in this identifier. In contrast, its feature *Size* is 3 since there are three tokens in this identifier, including *get*, *Connection*, and *Result*.

ContainsVerb, *ContainsNoun*, and *ContainsAdj.* These three features check whether a specific identifier contains verbs, nouns, and adjectives. Some specific granularities of identifiers tend to

contain a verb (or verbs) (e.g., method names). In contrast, some granularities of identifiers usually contain a noun (or nouns) (e.g., global variable names). Hence, verbs, nouns, and adjectives may be good indicators for predicting those identifiers that need renaming. In this study, we employ the widely used Stanford POS tagger[7] to analyze the POS of each token in a specific identifier. By parsing the POS result of each token, we can obtain the binary values of these features.

Style. This feature measures the styles of identifiers—that is, camel case, snake case, and flat [10]. *Camel case* is the practice of defining identifiers in which each middle token begins with a capital letter without spaces or punctuation. *Snake case* is the practice of defining identifiers in which tokens are separated by one or more underscores ("_"). If a specific identifier does not follow the styles of camel case and snake case, it is regarded as having the style of *flat*. We propose this feature because different styles of identifiers may have different probabilities of being renamed. We check the cases of characters and the composition (especially the underscore "_") of an identifier to classify it into the three styles. For example, the identifier *trackChanged* has the style of camel case. The identifier *SORT_BY_YEAR* has the style of snake case. In contrast, the identifier *moveplaylist* has the style of flat.

Contains$. It is allowed to enclose the special symbol $ in identifiers in the Java programming language. However, the symbol $ is not suggested in identifiers [52]. Hence, a specific identifier with $ may have a larger probability of being renamed than other identifiers. We check whether the identifier contains $ to obtain the binary value of this feature by the lexical matching.

StartWithUpper. This feature checks whether the identifier begins with an uppercase character. Some specific identifiers are suggested to start with an uppercase character (e.g., class names), whereas others are not. Hence, the uppercase initial character may be a good indicator for predicting whether a specific identifier needs renaming or not. The value of this feature is easy to obtain, since we only need to analyze the case of the first character of a specific identifier to obtain the binary value of this feature.

DigitCount. This feature measures how many digits a specific identifier contains. Those identifiers that contain many digits are hard to understand. Hence, the number of digits in identifiers may be a good indicator for predicting identifiers that need renaming. By checking each character and summing up the number of digits, we can obtain the value of this feature.

Relation with Convention. The relation with convention feature group measures the relationship between identifiers and the corresponding code conventions. We believe that if the definition of a specific identifier does not follow the common code convention proposed by the programming language or the common code convention widely recognized by developers, then this identifier has a high probability of being renamed [2, 12]. Hence, such a relationship could help us identify which identifiers need renaming. There are four features in this group, including *FollowConPOS*, *FollowConStyle*, *FollowCommonPOS*, and *FollowCommonStyle*. We check whether identifiers follow two code conventions (i.e., POS and Style) proposed from two aspects: the programming language and developers' common practice.

FollowConPOS and *FollowConStyle* check whether the POS and the style of a specific identifier follow the corresponding code conventions. First, we manually read and summarize the defined code conventions by the Java programming language.[8] By reading the rules of naming conventions for different granularities of identifiers, we translate them into a series of heuristic rules. For example, the code conventions defined by the Java programming language say "Class names should be nouns, in mixed case with the first letter of each internal word capitalized." Hence, we can find a heuristic rule for the POS of a class name, namely at least one of the substitutive tokens

[7] https://nlp.stanford.edu/software/tagger.html.
[8] https://www.oracle.com/java/technologies/javase/codeconventions-namingconventions.html.

in the class names shall be nouns. In addition, we can define a heuristic rule for the style of a class name—that is, the style of a class name should be the *upper camel case* style. When calculating the features involving code conventions, we check whether a specific identifier follows its corresponding heuristic rules. If it follows, this feature will be *true*. Otherwise, this feature will be *false*. In such a way, we can mine the code conventions and obtain the two features by checking the heuristic rules mined from code conventions. For example, the class name *CrashlyticsWrapper* follows the style defined in the code convention (i.e., the upper camel case style). In contrast, the class name *trackInfoUpdate* does not follow the style defined in the code convention. Hence, the values of the feature *FollowConStyle* are true for the first class name and false for the second class name.

In addition, we obtain the code conventions that are commonly recognized by developers for defining identifiers by mining and summarizing all the projects in the dataset. We extract the POS and style of each identifier and calculate the frequency of POS and style for each granularity of identifiers. We rank the POS and style based on their frequencies and select the top-ranked POS and style (when the accumulated percentage reaches 80%) as the commonly used code conventions adopted by developers. After obtaining the commonly used POS and style for each granularity of identifiers, *FollowCommonPOS* and *FollowCommonStyle* examine whether a specific identifier follows the obtained commonly used POS and style adopted by developers. If the identifier follows the corresponding POS or style, this feature will be true. Otherwise, this feature will be false.

Relation with Code Entities. The features in the relation with code entities group measure the relationship between identifiers and relevant code entities in the same project [16, 40]. Intuitively, if an identifier has the lexical or semantic relationship with other code entities, the identifier and the related code entities tend to be renamed or not simultaneously [22]. Hence, we can utilize the renaming information of related code entities to detect those identifiers that need renaming. In total, this feature group contains seven features, as follows.

The features *ContainOthers* and *ContainedByOthers* measure whether a specific identifier contains or is contained by other code entities in the same source file. If an identifier contains or is contained by other code entities, they may have a strong lexical and semantic relationship with each other. Hence, they tend to be renamed or not simultaneously. For example, it is not uncommon that there is a global variable *size* in a source file and there are two methods of *setsize* (the *setter* method) and *getsize* (the *getter* method) in the same source file, where the main functions of the setsize and getsize methods are to set and get the value of the global variable *size*. In this example, the identifier *size* is contained by setsize and getsize. If the global variable *size* is renamed, the getter and setter methods shall also be renamed. For the calculation of the two features, we extract all the other identifiers in the same source file in the historical version and perform the lexical matching between the specific identifier and the other identifiers. In such a way, we can obtain the binary values (contain or not contain, is contained by or is not contained by) of the two features.

The two features *SameNameSameGra* and *SameNameDiffGra* measure whether there are identifiers with the same name as the specific identifier in the same granularity and in a different granularity in the whole project, respectively. Similar to *ContainOthers* and *ContainedByOthers*, the rationales behind *SameNameSameGra* and *SameNameDiffGra* are that if two identifiers have the same name either in the same granularity or in a different granularity, they have relatively high probabilities to be renamed or not simultaneously. The same name of identifiers may be caused by code clone or some code recommendation techniques used by developers. The values of the two features are binary values. We extract all the other identifiers with their corresponding granularities in the same project and perform the lexical matching with the specific identifier for both the name and the granularity to obtain the values of the two features.

The feature *ContainFather* checks whether the specific identifier contains its "father" identifier or not. In the Java programming language, different granularities of identifiers have different scopes. We define the exact identifier whose scope contains this specific identifier as its "father." For example, the "father" identifier of a local variable is its corresponding enclosing method, and the "father" identifier of a method is its enclosing class. When developers define "child" identifiers, they usually add additional tokens on their "father" identifiers to make them concrete. The rationale behind this feature is that once a "father" or a "child" identifier is renamed, its corresponding "child" or "father" identifier also has a higher probability of being renamed. Similar to the preceding features, we can use the lexical matching between a "father" identifier and a "child" identifier to obtain the exact binary value of this feature. For example, a method *ExpressionImplement* is defined in the class *Expression*. We can see that the method name *ExpressionImplement* contains the class name *Expression*, so the value of this feature *ContainFather* is true for this method.

The feature *MinEdit* measures the minimal edit distance between the specific identifier and other identifiers in the same project. The edit distance is measured by counting the least number of operations required to transform an identifier to another identifier. It is also a widely used similarity metric. Similar to the preceding features, the less edit distance between the two identifiers, the more similar are the two identifiers, and the more chances that the two identifiers are renamed or not simultaneously. When we calculate the value of this feature, we extract all the identifiers in the historical version and compute the edit distance between the specific identifier and other identifiers. We select the minimal value as the final value of this feature.

The feature *MaxJaccard* is similar to the feature *MinEdit*. The only difference is that we employ the Jaccard similarity to calculate the similarity between two identifiers and select the maximal value as the final value of this feature. When we calculate the Jaccard similarity between two identifiers, we first split the two identifiers into two sets of composing tokens, respectively, using INTT. After we obtain the two sets of composing tokens for the two identifiers, we calculate the size of the intersection divided by the size of the union of the two sets as the Jaccard similarity. We select the maximal value of the Jaccard similarity between the specific identifier and other identifiers as the value of this feature.

Enclosing File. The enclosing file feature group measures the properties of the enclosing file for the current identifier. We believe that developers may modify the enclosing file and the specific identifier at the same time. Hence, the enclosing file and identifiers may evolve simultaneously, so measuring the properties of enclosing file could help us predict identifiers that need renaming [15]. We extract five features from the enclosing file as follows.

LOC. This feature measures the **Lines of Code (LOC)** of the enclosing file. Furthermore, *ImportCount* and *CommentCount* check the number of *import* statements and comments, respectively, in the enclosing file.

ConditionCount and *IterationCount*. These features check the exact number of conditional statements (*if* statement and *switch* statement) and iteration statements (*for* statement and *while* statement) in the enclosing file, respectively. The more conditional (iteration) statements defined in the enclosing file, the more complex is the enclosing file. Hence, to make it more clear and logical, developers may rename some identifiers.

History. The history feature group detects the change history (especially the change count) of the identifier itself, the enclosing statement, and the enclosing file. By digging the change history, we may find some clues as to which identifiers will need renaming in the future [63]. In detail, we design three features in this group. *ChangeCount*, *StateChangeCount*, and *FileChangeCount* measure the change count of the identifier itself, the enclosing statement, and the enclosing file in history, respectively. The more times the identifier and its programming context are changed, the higher the probability that the identifier is renamed in the future, since it may be hard to choose a

suitable name for the identifier. By analyzing the change history of the identifier and its programming context, we can obtain the exact values of these features.

Different groups of features measure and capture the naturalness of the renaming of identifiers. The exact values of these features can be easily obtained by performing static code analysis [54]. These features cooperate and complement to each other to help us pick out those identifiers that need renaming. The contribution of these features to the prediction of identifier renaming will be evaluated in Section 5.

3.1.3 Classifier Training. In this study, we employ the widely used Random Forest classifier as the default classifier, due to two reasons. First, it is one of the most widely used classifiers in software engineering. The Random Forest classifier is easy to use, and it could usually achieve satisfactory classification results. Second, a lot of similar studies also employ the Random Forest classifier by default without tuning its parameters [6, 8, 61]. For example, Aniche et al. [6] conducted extensive empirical experiments to prove that the Random Forest classifier is the best classifier for predicting software refactoring opportunities, including class renaming opportunities, method renaming opportunities, and local as well as global variable renaming opportunities. Similarly, one aim of our study is also to predict the renaming opportunities for all the granularities of identifiers. Hence, driven by the two reasons, we also employ the Random Forest classifier in this study. The classifier is an important component in the prediction phase of our approach. It has a relatively large impact on the performance of our approach. Selecting a good classifier could improve the performance of our approach. On the contrary, choosing a bad one could decrease its performance. Hence, it is necessary to figure out whether employing the Random Forest classifier as the default one is effective or not. In Section 5, we will validate the effectiveness of the Random Forest classifier.

We employ the package in WEKA,[9] a popular machine learning software in Java, to help us implement the Random Forest classifier. There are several parameters (e.g., the number of trees and the maximum depth of the trees) in the Random Forest classifier, and we do not propose an optimizer or a method to optimize these parameters in this classifier. This means that we leave these parameters as their default values in WEKA. For example, the maximum depth of the trees is unlimited. We clearly know that by elaborately tuning these parameters in the Random Forest classifier, our prediction results could be further improved. We make such a decision because we want to better show the potential of the Random Forest classifier. In addition, we want the trained model to be more reusable so that it can be easily reused, generalized, and adapted to different situations.

After we extract the five groups of features from each identifier, we combine them with the corresponding class label (i.e., whether the identifier needs to be renamed or not). Based on the training set, we can train a classifier. The Random Forest classifier works basically four steps:

(1) Select random samples from the given training instances.
(2) Construct a decision tree for each sample and obtain a prediction result based on each decision tree.
(3) Perform a vote for each predicted result.
(4) Select the prediction result with the most votes as the final prediction.

After the Random Forest classifier is fully trained, it can be used to predict whether a specific identifier needs to be renamed or not. Specifically, given a new piece of source code, we first use the Identifier Extractor component to extract all the granularities of contained identifiers. Then,

[9] https://www.cs.waikato.ac.nz/ml/weka/.

we employ our Feature Extractor component to extract all the defined features for each identifier. Finally, taking these features as input, the trained Random Forest can be used to predict whether a specific identifier needs to be renamed or not. Once an identifier is judged as needing renaming, we further suggest a new identifier list to help developers replace the flawed identifier to accelerate the identifier renaming process.

3.2 The Renaming Suggestion Phase

The aim of the renaming suggestion phase is to automatically provide a new identifier list for those identifiers needing renaming. As a whole, this phase contains six components, including Revision Pattern Identification, Pattern-Based Scoring, Identifier Mapping Construction, History-Based Scoring, Final Score Calculation, and Candidate Ranking. We explain the details of these components as follows.

3.2.1 Revision Pattern Identification. This component aims to identify a series of revision (renaming) patterns of identifiers from the training set. This component is specially designed for extracting revision patterns of identifiers at the token level, which is the first work to employ the revision patterns as far as we know. The most similar studies only proposed the syntactical patterns of identifiers from the POS perspective [39], which are different from the revision patterns in this study. Since we can obtain the exact identifiers needing renaming and their corresponding correct new identifiers in the training set, we mine the revision patterns based on the modifications from the flawed old identifiers to the correct new identifiers. Specifically, we propose a revision pattern identification algorithm as shown in Algorithm 1. This algorithm takes in a set of identifier changes in the training set $\{OI_1 \rightarrow NI_1, OI_2 \rightarrow NI_2, \ldots, OI_N \rightarrow NI_N\}$, where OI is an old identifier and NI is the corresponding new identifier. As the output, the algorithm generates a set of revision patterns $P = \{P_1, P_2, \ldots, P_M\}$, where $p_i = O_i \rightarrow N_i$. These revision patterns are expressed in the form of $O_i \rightarrow N_i$, which means that the old token O_i is changed to the new token N_i (O_i or N_i can be empty). In addition, the revision patterns include the change revision, the deletion revision, and the addition revision.

First, we tokenize the old identifier as well as the new identifier into a set of old tokens $O = \{O_1, O_2, \ldots, O_M\}$ and a set of new tokens $N = \{N_1, N_2, \ldots, N_K\}$, respectively, by using INTT. After both the old identifier and the new identifier are tokenized, we remove the overlapping tokens that appear in both the sets of old tokens and new tokens, since these constitutive tokens are not changed when identifiers are renamed. It means that these tokens may well represent the lexical and semantic meaning of identifiers. For the remaining tokens in the set of old tokens and new tokens, if both of the two remaining sets are not empty, we generate a series of change revisions and count their frequencies. We iterate all the renaming tokens in O and N and generate $|O| \times |N|$ change revisions as $\{O_1 \rightarrow N_1, \ldots, O_1 \rightarrow N_{|N|}, O_2 \rightarrow N_1, \ldots, O_2 \rightarrow N_{|N|}, \ldots, O_{|O|} \rightarrow N_1, \ldots, O_{|O|} \rightarrow N_{|N|}\}$. For each of the change revision, we add it into the set of the revision patterns and increase its frequency. By default, the frequency of a newly generated change revision that does not exist in the set of the revision patterns is 0.

If the remaining set of old tokens is not empty but the remaining set of new tokens is empty, it means that these old tokens are removed when identifiers are renamed. In such a situation, we generate a series of deletion revisions and count their frequencies. Specifically, we generate $|O|$ deletion revisions as $\{O_1 \rightarrow \Phi, O_2 \rightarrow \Phi, \ldots, O_{|O|} \rightarrow \Phi\}$, where Φ means empty. Similarly, we iteratively add each generated deletion revision into the set of the revision patterns and increase its frequency.

If the remaining set of old tokens is empty but the remaining set of new tokens is not empty, it means that these new tokens are added when identifiers are renamed. In such a situation, we generate a series of addition revision and count their frequencies. Specifically, we generate $|N|$

ALGORITHM 1: The Revision Pattern Identification Algorithm

Input: A set of identifier changes in the training set $\{OI_1 \rightarrow NI_1, OI_2 \rightarrow NI_2, \ldots, OI_N \rightarrow NI_N\}$, where OI is an old identifier and NI is the corresponding new identifier
Output: A set of revision patterns $P = \{P_1, P_2, \ldots, P_M\}$, where $p_i = O_i \rightarrow N_i$

```
while all identifier changes OI_i → NI_i do
    Tokenize OI_i into a set of tokens O = {O_1, O_2, ..., O_M};
    Tokenize NI_i into a set of tokens N = {N_1, N_2, ..., N_K};
    Remove overlapping tokens in both O and N;
    if O! = Φ and N! = Φ then
        for i = 1 to |O| and j = 1 to |N| do
            generate a change pattern P_ij = O_i → N_j;
            add P_ij to P;
            frequency(P_ij) ++;
        end
    else if O! = Φ and N = Φ then
        for i = 1 to |O| do
            generate a deletion revision P_i = O_i → Φ;
            add P_i to P;
            frequency(P_i) ++;
        end
    else if O = Φ and N! = Φ then
        for j = 1 to |N| do
            generate an addition revision P_j = Φ → N_j;
            add P_j to P;
            frequency(P_j) ++;
        end
    else
        continue;
    end
end
```

addition revisions as $\{\Phi \rightarrow N_1, \Phi \rightarrow N_2, \ldots, \Phi \rightarrow N_{|N|}\}$. We then continuously add each generated addition revision into the set of the revision patterns and increase its corresponding frequency.

When we check all the pairs of flawed old identifiers and corresponding correct new identifiers, we accumulate the frequencies of generated revision patterns. Intuitively, the higher the frequency of a revision pattern, the more likely the revision pattern is applied to those identifiers need renaming, since relevant identifiers tend to show similar renaming operations. Based on these revision patterns, we can generate a series of new identifiers by modifying the old identifiers. For example, we find that the constitutive token *btn* is usually changed into *button* in identifiers to perform the abbreviation expansion, which is a revision pattern and can be expressed as *btn* → *button*. Hence, when an identifier that needs renaming contains the constitutive token *btn*, it is likely that the constitutive token *btn* is changed to *button* to form the new identifier by applying the revision pattern. As another example, we find that the constitutive token *value* is often deleted in identifiers when they are renamed, possibly because this token contains less information. This deletion revision can be expressed as *value* → Φ. Hence, once an identifier needing renaming contains the constitutive token *value*, it is possible that this token is deleted from the identifier.

3.2.2 Pattern-Based Scoring. The aim of the pattern-based scoring is to form a set of suggested new identifiers and calculate their relevant scores for the identifiers that need renaming based on the generated revision patterns, which is a novel component designed in this study.

Given a current identifier CI that needs renaming, we first leverage INTT to tokenize it into a set of constitutive tokens $\{T_1, T_2, \ldots, T_N\}$. Then, based on the revision patterns, we perform a series of mutation operations to CI to generate a set of new identifiers, including change mutation, deletion mutation, and addition mutation.

Change Mutation. For each composing token T_i, we examine the generated revision patterns to find whether a specific change pattern contains it. If the old token in a change pattern is the same as this token T_i, we replace T_i with the corresponding new token in the change pattern. In such a way, by checking all the composing tokens and replacing them with new tokens, we can generate a series of new identifiers. In addition, we record the frequency of each applied change pattern and use this frequency to present the relevance of the corresponding generated new identifier. The higher the frequency, the higher probability that CI is changed to the generated new identifier.

Deletion Mutation. For each composing token T_i, we check whether a specific deletion revision contains T_i. If there exists a deletion pattern contains T_i, then we delete T_i from CI to form a new identifier. Similarly, we also record the frequency of the applied deletion revision, since a higher frequency usually indicates a larger probability of the generated new identifier.

Addition Mutation. Before we apply the addition mutation, we first obtain the common addition revision in the same granularity as CI. We believe that identifiers in the same granularity may add the same or similar tokens when they are renamed. We rank the addition revision based on their frequencies and select the top 10 addition revisions to apply. For these 10 added tokens in the addition revision, we insert them into CI one at a time. In this study, we try all the possible locations to insert these added tokens, including the front, the middle, and the end of CI. When we insert the added token in the middle of CI, we can only insert it between the composing tokens rather than all the inner locations of the composing tokens. This means that we cannot break up existing composing tokens when performing the addition mutation in the middle. In such a way, we can generate 10 new identifiers.

After we perform a series of change, deletion, and addition revisions to the current identifier CI that needs renaming, we can generate a set of new identifiers $\{NI_1, NI_2, \ldots, NI_N\}$. For each of the newly generated identifiers NI_i, we need to calculate a relevant score for it so that we can rank all of them. Since we have already recorded the frequency of each applied revision pattern, based on the corresponding frequency, we calculate the relevant score between CI and NI_I as follows.

$$Relevance1(CI, NI_i) = \frac{Frequency\ of\ applied\ pattern}{\sum Frequencies\ of\ all\ applied\ patterns} \tag{1}$$

3.2.3 Identifier Mapping Construction. The identifier mapping construction component aims to map the current flawed identifier with new correct identifiers in the training set by taking similar identifiers as intermediaries. We design it based on the change history of identifiers by ourselves.

Specifically, we propose an identifier mapping construction algorithm as shown in Algorithm 2, which takes the set of identifier renaming changes in the training set $\{OI_1 \rightarrow NI_1, OI_2 \rightarrow NI_2, \ldots, OI_N \rightarrow NI_N\}$ and the current identifier CI that needs renaming as input and outputs the mappings between the current identifier CI and the set of relevant new identifiers NI.

As shown in Algorithm 2, we first define a set of all new identifiers AI, which is empty by default. Then, we check all the identifier renaming changes in the training set. If a specific new identifier NI_i in the identifier renaming change is not contained in AI, we add NI_i into AI. In such a way, we can obtain all the unique new identifiers in the training set. Then, we further define an empty set of relevant new identifiers NI. For each identifier NI_i in AI, we check whether there exists OI_i, whose similarity with CI is larger than 0. If it is true, we construct a mapping between CI and NI_i, and add NI_i into the set NI. After checking all the new identifiers in AI, we can generate a set of relevant new identifiers for CI.

ALGORITHM 2: The Identifier Mapping Construction Algorithm

Input: A set of identifier changes in the training set $\{OI_1 \rightarrow NI_1, OI_2 \rightarrow NI_2, \ldots, OI_N \rightarrow NI_N\}$, where OI is an old identifier and NI is the corresponding new identifier in the training set; the current identifier CI that needs renaming

Output: The mappings between the CI and all the relevant new identifiers $NI = \{NI_1, NI_2, \ldots\}$

```
the set of all new identifiers AI = Φ;
while all the identifier changes in the training set do
    select a new identifier NI_i from a identifier change;
    if NI_i ∉ AI then
        add NI_i into AI;
    end
end
the set of relevant new identifiers NI = Φ;
while all the new identifiers in AI do
    select a new identifier NI_i from AI;
    if exist(similarity(OI_i of NI_i, CI) > 0) then
        construct a mapping between CI and NI_i;
        add NI_i into NI;
    end
end
```

In this study, we employ the Jaccard similarity to calculate the similarity between CI and OI_I, since the Jaccard similarity is a widely used measurement that can measure the similarity at the token level rather than the character level. Specifically, we split both CI and OI_i into two sets of tokens with INTT. Jaccard similarity is calculated by the number of tokens in both sets divided by the number of tokens in either set. The Jaccard similarity ranges from 0 to 1. The higher value of the Jaccard similarity, the more similar are CI and OI_i.

3.2.4 History-Based Scoring. The aim of the history-based scoring component is to calculate the relevant scores between the current identifier CI that needs renaming and all the mapped new identifiers, whose basic structure is derived from a similar work related to bug localization [64]. Specifically, based on the output the identifier mapping construction algorithm shown in Algorithm 2, we can form the mapping graph as shown in Figure 6. This mapping graph contains three layers. The first layer only includes the current identifier CI that needs renaming. The second layer of the graph contains the similar old identifiers OI in the training set. The links between the the first layer and the second layer indicate the Jaccard similarity between CI and OI_i with a value larger than 0. The third layer shows all the corresponding new identifiers NI_i for the OI_i in the second layer, and the links demonstrate the exact identifier renaming change.

For each NI, we can calculate its relevant score with CI as follows.

$$Relevance2(CI, NI_i) = \sum_{all\ OI_k\ linking\ to\ NI_i} Jaccard(CI, OI_k) \tag{2}$$

In this study, we employ the Jaccard similarity to calculate the similarity between CI and OI_k. Based on Formula (2), we can calculate a relevant score for each new identifier.

3.2.5 Final Score Calculation. Based on the change patterns and identifier mappings, we can obtain two sets of newly generated identifiers with their relevant scores for each identifier CI that needs renaming. The final score calculation component aims to merge the two sets of newly generated identifiers and calculate their final scores.

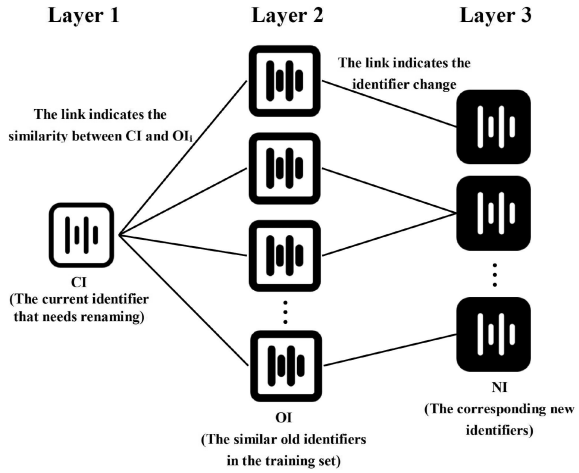


Fig. 6. The mappings between the current identifier and new identifiers.

Specifically, some of the newly generated identifiers emerge in both sets, whereas the others only appear in one set. Hence, we first merge the two sets together to form a whole set that contains all the unique identifiers. Then, for each unique generated identifier NI_i, we normalize the scores obtained by Formula (1) and Formula (2) to the range between 0 and 1. Finally, we calculate its final relevant score as follows:

$$Final(CI, NI_i) = Relevance1(CI, NI_i) + Relevance2(CI, NI_i), \tag{3}$$

where $Relevance1$ is the normalized result of Formula (1) and $Relevance2$ is the normalized result of Formula (2). It should be noted that the result of one relevant score may be 0 for some newly generated identifiers. For example, given that the new identifier *getwarning* is generated from both the renaming history and the renaming patterns, whose normalized relevant scores are 0.64 and 0.48, respectively, our approach calculates the final score for this new identifier as 0.64 + 0.48 = 1.12. As another example, the new identifier *createClassField* is only generated from the renaming patterns of our approach, whose relevant score is 0.72. However, this identifier cannot be obtained or generated from the renaming history. In such a situation, the relevant score of this identifier from the renaming history is 0. Hence, the final score of this new identifier is only 0.72 + 0 = 0.72 calculated by our approach.

In this study, we simply give the two relevant scores the same weight, due to two reasons. First, giving the same weight makes our approach simple and easy to use in practice, which could show the great potential of our approach. We certainly can further improve the performance of our approach by elaborately adjusting and tuning different weights to the results of Formula (1) and Formula (2). Second, this decision makes our approach easy to generalize and adapt to the new scenarios. When our approach is evaluated over a new project or dataset, we cannot figure out the exact values of the weights that help our approach achieve the best results in advance. Hence, in the new scenario, we can also use the same weight without having to adjust them. Users will find that it is easy to apply our approach in a different scenario.

3.2.6 Candidate Ranking. Based on the final scores, we rank the newly generated identifiers and select the top 10 identifiers to recommend for each identifier that needs renaming. Suggesting 10 results is commonly adopted by recommendation systems in software engineering, since the suggestion list is not too long and scanning the suggestion list with such a length will not cost

much time for developers. In this study, we evaluate the recommendation performance based on the 10 results with a notation @K, where K ranges from 1 to 10. We provide the suggestion list for developers so that they can pick up one suitable identifier from it to replace the old identifier that needs renaming. In such a way, our approach could accelerate the identifier renaming and refactoring activities for developers.

4 EXPERIMENTAL SETUP

In this section, we present the details of the experimental data and its preliminary exploration, the investigated **Research Question (RQs)**, the state-of-the-art baseline approach that we want to compare, the evaluation method, and evaluation metrics.

4.1 Data Collection and Preliminary Exploration

In this section, we show how we collect the experimental projects, how we construct the ground truth for identifier renaming prediction and suggestion, and some preliminary exploration on the identifier renaming.

4.1.1 Data Collection. In this work, we collect two categories of open source projects hosted in GitHub: the Java projects and the Android projects. We validate our approach over the two categories of projects, since we want to figure out whether our approach performs differently or similarly over the two project categories. If our approach achieves similar results over the two project categories, we may say that our approach is insensitive to the project categories.

Specifically, we collect 100 open source projects in total, with 50 Java projects and 50 Android projects. Compared against existing related studies (e.g., Liu et al. [39], who conduct experiments only on four popular projects), the number of our experimental projects is large enough. The reason we select the Java and Android projects is that the Java projects and the Android projects have a lot in common. Most of them are developed by the Java programming language. Therefore, when we design our approach, we can uniquely design the same steps for data processing. In addition, the Java projects and the Android projects run on different platforms, making their development methods very different. For example, the update frequency of the Java projects is slower than that of the Android projects, which may result in more identifiers that need renaming. The difference in development methods affects the characteristics and the quality of the experimental projects to a certain extent, which in turn affects the performance of our proposed approach. Hence, we select the two types of projects and hope that the proposed approach performs similarly on the two types of projects.

These collected 100 projects are derived from a publicly available dataset published by Allamanis and Sutton [5]. This dataset contains software projects with different domains, popularity, and sizes. We pick up the target experimental projects from this dataset. When selecting the experimental projects, we do not want to only select the widely used popular projects. Popularity projects may contain sufficient historical code changes and relatively good identifiers, which introduces bias to (especially improve) the performance of different approaches. Hence, we select experimental projects from the datasets with different popularity to better simulate the real scenario. To achieve this, we randomly select 50 Java projects and 50 Android projects, respectively. For the 50 Java projects, they have different degrees of popularity and code sizes from different domains, such as library and framework. For the 50 Android projects, they have different functions and types, such as Finance and Wallpaper. We believe that such a variety of collected projects could make a fair evaluation to different approaches.

Table 3 summarizes the average values of main characteristics of the two categories of projects. Among them, *Watcher*, *Star*, and *Fork* are usually regarded as the popularity indicators for projects.

Table 3. Average Values of Main Characteristics of Experimental Projects

Project	Watcher	Star	Fork	Request	Commit	Branch	Release	Contributor	File	LOC	Identifiers
Java	20.52	173.72	73.4	4.08	1,074.34	9.44	20.12	8.72	530.2	89,167.28	15,909.72
Android	49.16	678.62	217.42	2.36	1306.7	3.94	9.76	8.42	149.22	24,537.7	5,466.14

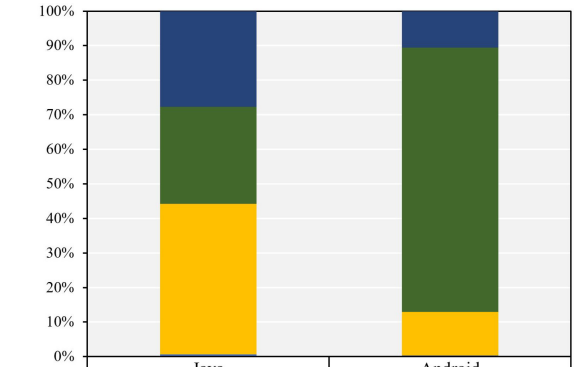
From these characteristics, we can see that the average numbers of *Watcher*, *Star*, and *Fork* of the Android projects are larger than those of the Java projects. This means that the Android projects receive more attention from developers and users, since Android developers are booming today and the Android projects tend to be developed based on existing product lines. In addition, *Request*, *Commit*, *Branch*, *Release*, and *Contributor* are often used as the indicators of development activities conducted by developers. From these characteristics, we can find that except for *Commit*, the average values of the other characteristics of the Java projects are larger than those of the Android projects. This means that the Java projects accumulate more historical development activities than the Android projects. The rest, *File*, *LOC*, and *Identifiers*, are usually used to evaluate the sizes of projects. We can see that the Java projects contains more files, LOC, and identifiers than the Android projects. This may be because the Java projects usually provide more functions than the Android projects.

4.1.2 Ground Truth Construction. The aim of this study is to automatically predict which identifiers need renaming and further provide a new identifier suggestion list for developers to replace them. Thus, it is impossible for us to do such things based on the current version of the experimental projects, since we could not know which identifiers need renaming in the future and what the target correct identifiers are. As a result, we need to trace back to a historical version and conduct the identifier renaming prediction and suggestion based on the historical version for these experimental projects. Specifically, we set up the time interval to half a year, which means that we use the "git reset" command to roll back to the historical version with no less than 6 months. In such a way, we can obtain the two versions of each experimental project: a historical version and the current version. By parsing and comparing each source file in the historical version and the corresponding one in the latest version, we could figure out which identifiers have been renamed. We regard those identifiers that have been renamed as positive instances and the other identifiers that are not changed at all as negative instances. We conduct all the experiments on the identifiers extracted from the historical version of the experimental projects and regard the corresponding ones in the current version as the target correct identifiers. In such a way, we can construct a ground truth dataset containing the exact identifiers needing renaming and their corresponding correct target identifiers.

4.1.3 Preliminary Exploration.

Preliminary Exploration Constructs. After we collect the experimental projects and construct the ground truth, we continue to explore the naturalness of identifiers that need renaming by performing some preliminary explorations. Specifically, we investigate the naturalness of identifiers that need renaming from three dimensions, including the distribution of identifiers that need renaming over different granularities, the length disparity, and style changes of identifiers before and after renaming:

- In terms of the first dimension, exploring the distribution of identifiers that need renaming over different granularities could help us figure out which specific granularity of identifiers tends to be (not) renamed and further shed light on the design of our proposed approach as well as the future research. We calculate the percentage of each granularity of identifiers that need renaming to show the corresponding distribution.



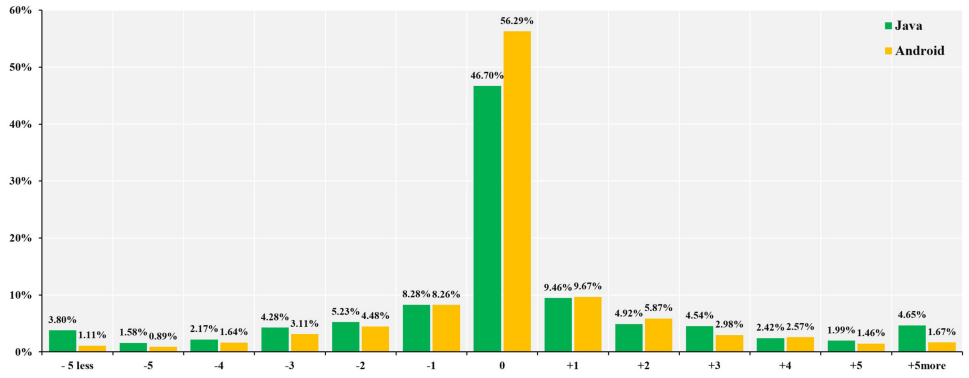
	Java	Android
Local	27.73%	10.62%
Global	28.01%	76.46%
Method	43.62%	12.53%
Type	0.63%	0.39%
Package	0.01%	0.00%

Fig. 7. The distribution of identifiers that need renaming in different granularities.

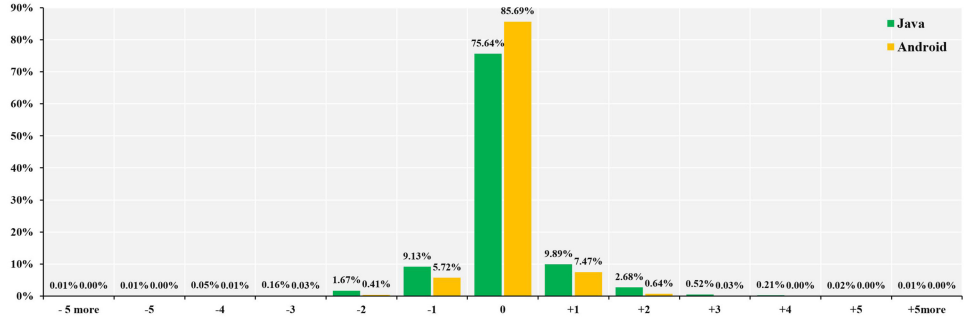
- As for the second dimension, investigating the differences of the lengths of identifiers before and after renaming could help us get some inspiration around how to generate new identifiers for those that need renaming, especially how to determine the lengths of new identifiers. Identifiers can be divided into both character level and token level. To provide a more comprehensive overview, we calculate the result of the length of an identifier before renaming minus the length of the identifier after renaming for all the identifiers that need renaming and calculate the corresponding percentage at both the character and token levels.
- In terms of the third dimension, we explore the styles of identifiers before and after renaming. Such an investigation could help us better determine the exact style when renaming identifiers. If we want to suggest the exact new identifiers for those needing renaming, one of the sub-tasks is that we should define the styles of new identifiers. Only if we can know the styles of new identifiers can we suggest correct new identifiers. We compare the styles of identifiers before and after renaming and report the style changes as well as their corresponding percentages.

Preliminary Exploration Results. Figure 7 presents the results of the first dimension—that is, the distribution of identifiers that need renaming in different granularities. We can see that the Java projects and the Android projects show different trends in terms of the distribution of identifiers that need renaming. For the Java projects, more than 43% of identifiers needing renaming are method names. This shows that method names are easy to be renamed in the Java projects. The percentages of global variables and local variables are similar (i.e., 28.01% and 27.73%). This means that global variables and local variables may perform similarly in terms of renaming. As for package names and type names, they are seldom renamed in the Java projects. In contrast, more than three-fourths of identifiers that need renaming are global variables in the Android projects. This may be because Android developers usually define a lot of global variables to hold some shared values. In addition, 10.62% and 12.53% of identifiers that need renaming are local variables and method names, respectively. Similarly, package names and type names are also seldom renamed.

Figure 8 shows the disparity of the lengths of identifiers before and after renaming at the character level and token level, respectively. Such a disparity is calculated by the length of an identifier before renaming minus the length of the identifier after renaming. Hence, the disparity can be a



(a) The length disparity of identifiers before and after renaming at the character level.



(b) The length disparity of identifiers before and after renaming at the token level.

Fig. 8. The length disparity of identifiers before and after renaming in the Java and Android projects.

positive value or a negative value. We can see that the length of 46.70% and 56.29% of identifiers are not changed at the character level in the Java projects and the Android projects, respectively. This shows that even though identifiers are renamed, the length of about 50% of these identifiers are exactly the same. Along with the increase of the disparity (no matter negative or positive) of the length before and after identifier renaming, the corresponding percentage shows a downward trend. The disparity at the token level is similar to that at the character level—that is, the disparity with no change takes up the largest percentage, and along with the increase of the disparity, the percentage presents an even rapid downward trend at the token level.

Table 4 summarizes the style changes of identifiers before and after renaming in both the Java and Android projects. We can see that even though identifiers are renamed, their corresponding styles remain the same with a percentage of more than 95% in the Java and Android projects. In addition, the transactions among camel case, snake case, and flat are seldom for identifiers that need renaming. This finding is important to the design of our approach. When we generate new identifiers for those needing renaming, we just need to follow the same old style of these identifiers.

4.2 Research Questions

In this section, we illustrate the investigated RQs that we want to answer in this study. Specifically, we answer six RQs, and they can be divided into two parts. The first part includes four RQs that aim to evaluate the performance of our approach, as well as its main components in predicting identifiers needing renaming. Meanwhile, the second part includes two RQs to evaluate the

Table 4. Style Changes of Identifiers Before and After Renaming

Style Change	Java	Android
Unchanged	95.87%	95.10%
Snake → Camel	0.36%	0.30%
Snake → Flat	0.29%	0.83%
Camel → Snake	0.36%	0.27%
Camel → Flat	1.35%	1.10%
Flat → Snake	0.29%	1.17%
Flat → Camel	1.49%	1.23%

performance of our approach and its components in suggesting new identifiers. The detailed RQs are shown as follows.

RQ1: To What Extent Is Our Approach Superior to the Existing State-of-the-Art Approach in Predicting Identifiers That Need Renaming? For this RQ, we explore whether and to what extent our approach outperforms the state-of-the-art approach in predicting identifiers needing renaming. In this RQ, we investigate the construct of the performance of our approach in predicting identifiers that need renaming. The performance is measured by how accurately (measured by Precision) and completely (measured by Recall) our approach can predict identifiers that need renaming. In particular, we reproduce *DeepMethod* and *RenameExpander* for comparison, which are the state-of-the-art approaches in the literature. We validate both our approach and the state-of-the-art baseline approaches on the experimental dataset and calculate the values of Precision, Recall, and F-Measure to compare the performance of our approach and the baseline approaches.

RQ2: What Is the Influence of the Proposed Groups of Features to the Prediction Results of Our Approach? For this RQ, we investigate the construct of the performance of the proposed features in our approach. Similar to the previous experiment, we employ Precision, Recall, and F-Measure as the evaluation metrics. The performance of the proposed features is also measured by how accurately and completely different groups and combinations of groups of features can help our approach in predicting identifiers that need renaming.

RQ3: Is the Default Random Forest Classifier Effective in Predicting Identifiers That Need Renaming? Similarly, for this RQ, we investigate the construct of the performance of the Random Forest classifier in our approach. To show the performance of the Random Forest classifier, we compare it against several other commonly used classifiers by measuring the achieved Precision, Recall, and F-Measure.

RQ4: What Is the Performance of Our Approach in the Context of Cross-Project Prediction? For this RQ, we investigate the construct of the performance of our approach in predicting identifiers that need renaming in the context of cross-project prediction. The same as the experiment in RQ1, we measure and compare the achieved Precision, Recall, and F-Measure of our approach and the baseline approaches in the context of cross-project prediction.

RQ5: Does Our Approach Achieve Better Results Than RefactorLearning in Suggesting Accurate Full Name Identifiers? For this RQ, we investigate the construct of the performance of our approach in suggesting new identifiers for those needing renaming. To measure the performance, we obtain the recommendation list of the new identifiers achieved by our approach and measure the ratio of the targeted identifiers that are correctly recommended in the recommendation list. In addition, we compare the performance (measured by the Hit Ratio) of our approach against that of the baseline approach *RefactorLearning* to show its effectiveness.

RQ6: Does Combining Both the Information of Renaming History and Renaming Patterns Help Improve the Performance of Our Approach in Suggesting New Identifiers? For

this RQ, we investigate the construct of the effectiveness of combining both the information of the renaming history and the renaming patterns in our approach. The final recommended new identifiers come from both the renaming history and the renaming patterns. The same as the experiment in RQ5, we compare our approach against its variants only using a specific source of information (either the renaming history or the renaming patterns) with the Hit Ratio as the evaluation metric.

4.3 Baselines

In this section, we present the details of the existing state-of-the-art baseline approaches. Specifically, to show the effectiveness of our proposed approach, we employ *DeepMethod* [36] and *RenameExpander* [39] as the baseline approaches for comparing the performance of predicting identifiers needing renaming. In addition, we employ *RefactorLearning* [41] as the baseline for comparing the performance of suggesting correct new identifiers. These baseline approaches have recently been proposed and typically achieve promising results.

DeepMethod is a deep learning based identifier renaming prediction approach [36]. This approach extracts and utilizes the historical code change information and the semantic relationships between identifiers. This approach includes three phases, including data pre-processing, model training, and renaming prediction and correction. In the data pre-processing phase, this approach employs an under-sampling strategy to balance the number of identifiers that need renaming and the number of identifiers that do not need renaming. Then, it tokenizes identifiers into several composing tokens based on the cases of characters and the underscore (_). In the model training phase, it first employs BERT (Bidirectional Encoder Representations from Transformers) to embed identifiers into vectors. Then, this approach uses textCNN to enhance these embedding vectors. These vectors are further input into the **Multilayer Perceptron (MLP)** classifier to learn the boundary between identifiers that need renaming and identifiers that do not need renaming. By minimizing the loss function, the model can be trained. In the renaming prediction and correction phase, the trained model can be used to predict whether a given new identifier needs renaming or not. Finally, the change histories of the semantically related identifiers are extracted to correct and refine the prediction results. Evaluated on only 10 project hosted in the Apache community, this approach achieves the average F-Measure of about 80%.

RenameExpander predicts those identifiers that need renaming by expanding conducted renaming [39]. The rationale behind this approach is that once an identifier renaming is conducted, the similar or closely related identifiers also need to be renamed. Hence, it is unnecessary to perform the difficult and complex source code semantic analysis and understanding for this approach. Specifically, this approach first detects a renaming refactoring from an old name to a new name for a specific identifier. Then, it analyzes the old name and the new name to generate an edit script, which could transform the old name to the new name with the smallest set of deletions, insertions, and replacements. Next, this approach searches for other closely related identifiers that are similar to the old name of the specific identifier. The similarity is measured by several ways, including inclusion, sibling, reference, and inheritance. If these similar identifiers satisfy some conditions, these identifiers are regarded as to be renamed. This approach is only evaluated on four open source projects. Experimental results show that this approach achieves an average Precision of 82%.

RefactorLearning includes two phases: the training phase and the identification & suggestion phase [41]. Originally, *RefactorLearning* was proposed targeted toward only method names. In this study, we expand it into the other granularities of identifiers. The rationale behind this approach is that identifiers should be semantically consistent with their contexts. If not, those identifiers should be renamed by suggesting a ranked list of new identifiers that are close to the contexts. In this study, the contexts of method names and class names are method bodies and class bodies. In addition, the contexts of global and local variables are their enclosed class bodies and method

Actual Class \ Predicted Class	Positive (Identifiers that need renaming)	Negative (Identifiers that do not need renaming)
Positive (Identifiers that need renaming)	**True Positive (TP)**	**False Negative (FN)**
Negative (Identifiers that do not need renaming)	**False Positive (FP)**	**True Negative (TN)**

Fig. 9. The structure of the confusion matrix with TP, TN, FP, and FN values.

bodies. Specifically, in the training phase, this approach employs the Paragraph Vector to embed identifiers and convolutional neural networks to embed the corresponding contexts. By minimizing the distance of similar identifiers and contexts in the embedding space, this approach can be trained. In the identification & suggestion phase, this approach searches for similar identifiers and contexts in the embedding space and ranks and recommends the identifier groups based on the average similarity and the size of the group for those inconsistent identifiers. Experimental results demonstrate that this approach achieves Hit@5 of 25.70% in suggesting new full name identifiers.

4.4 Evaluation Method

Similar to related studies, we evaluate our identifier renaming prediction and suggestion approach with the widely used 10-fold cross validation. For a specific software project, we first collect all the identifiers in this project. Next, we trace back to the historical version of this project so that we can figure out which exact identifiers need to be renamed and which identifiers do not need to be renamed. For the two categories of identifiers, we assign them with two class labels—that is, negative for those identifiers that do not need renaming and positive for those identifiers that need renaming. Then, we extract and calculate the value of each proposed feature for all the identifiers. In such a way, we can form a feature vector for each identifier and combine this feature vector with its class label (negative or positive) to represent each identifier. Next, we divide all the identifiers (represented by feature vectors and class labels) into 10 equally sized folders. Among the 10 folders, 9 folders are regarded as the training set to train the Random Forest classifier in our approach. The Random Forest classifier receives the feature vectors and the corresponding class labels and detects the boundary between the two categories of identifiers based on the training set. The remaining folder is regarded as the test set for evaluating the performance of our approach. Since there are 10 folders, we repeat and run this process 10 times. In each run, 1 unique folder is selected as the test set and the other 9 folders as the training set. Finally, we calculate the average value of the 10 runs as the final performance of our identifier renaming prediction and suggestion approach.

4.5 Evaluation Metrics

As we have described, there are two phases in our approach: renaming prediction and renaming suggestion. To better evaluate our approach, we employ different evaluation metrics to separately evaluate the two phases of our approach.

We regard renaming prediction as the typical classification task. Hence, we employ the widely used Precision, Recall, and F-measure to evaluate the performance of our approach in renaming prediction. These evaluation metrics are calculated based on the confusion matrix shown in Figure 9. Specifically, there are two classes in this study, including positives (i.e., identifiers that need renaming) and negatives (i.e., identifiers that do not need renaming), under two conditions: the actual class and the predicted class by our approach. As shown in Figure 9, an identifier needing renaming (positive) may be correctly predicted as needing renaming (positive) by our approach,

and we call the identifier a **True Positive (TP)**. In addition, an identifier needing renaming (positive) may be incorrectly predicted as not needing renaming (negative) by our approach, and we call the identifier a **False Negative (FN)**. For an identifier that does not need renaming (negative), our approach can also correctly predict it as not needing renaming (negative), and we call the identifier a **True Negative (TN)**. In contrast, it can also be incorrectly predicted as needing renaming (positive), and we call the identifier a **False Positive (FP)**. TP and TN are correctly predicted results by our approach, whereas FP and FN are incorrectly predicted results by our approach. Based on the confusion matrix, Precision, Recall, and F-measure can be calculated as follows.

$$Precision = \frac{\#TP}{\#TP + \#FP} \times 100\% \tag{4}$$

$$Recall = \frac{\#TP}{\#TP + \#FN} \times 100\% \tag{5}$$

$$F\text{-}measure = \frac{2 \times Precision \times Recall}{Precision + Recall} \times 100\% \tag{6}$$

For the renaming suggestion, we treat it as the typical recommendation task. Hence, we employ the widely used Hit Ratio as the evaluation metric. As mentioned previously, our approach will suggest a recommendation list with 10 ranked results. We check whether one specific suggested identifier is exactly the same as the target new identifier. If it is true, we call it a *Hit*. We then calculate the Hit Ratio as the percentage of identifiers that are correctly suggested in the recommendation list by our approach. We calculate the Hit Ratio from top 1 to top 10 with the notation Hit@K, where K ranges from 1 to 10. Hence, the Hit Ratio can be calculated as follows.

$$Hit@K = \frac{No.\ of\ correctly\ suggested\ identifiers\ in\ top\ K}{No.\ of\ all\ identifiers} \tag{7}$$

5 EXPERIMENTAL RESULTS

In this section, we conduct extensive experiments to validate the performance of our approach in predicting identifiers that need renaming and suggesting new identifiers for them. Hence, the evaluation includes two parts, namely renaming prediction and renaming suggestion.

5.1 Evaluation of Renaming Prediction

To better evaluate the results of renaming prediction, we investigate the following four RQs.

5.1.1 Prediction Performance Comparison.

RQ1: To What Extent Is Our Approach Superior to the Existing State-of-the-Art Approach in Predicting Identifiers That Need Renaming?

Motivation. To explore the effectiveness of our approach in predicting identifiers that need renaming, we compare it with the existing two state-of-the-art approaches, namely *DeepMethod* and *RenameExpander*. By investigating this RQ, we can also figure out whether the traditional classification method used by our approach is superior to the deep learning based approach. In this study, an approach is superior to the other approaches if it achieves better performance values of the employed evaluation metrics (e.g., Precision, Recall, and F-Measure) than the other approaches.

Approach. We re-implement the two state-of-the-art baseline approaches and validate our approach as well as the baselines in the Java and Android projects. All of these approaches are verified with the same evaluation method. By comparing Precision, Recall, and F-measure achieved by different approaches, we could know which approach is the best in predicting identifiers that need renaming. In addition, we conduct the statistical hypothesis test to check whether there are significant differences among the experimental results of the three approaches by applying the Wilcoxon

Rank Sum test with Bonferroni correction and Cliff Delta Effect Size. The Wilcoxon Rank Sum test is a non-parametric statistical test that compares two independent groups of observations. It does not make any assumptions about what kind of distribution is involved, including the normality of the data. Since the Wilcoxon Rank Sum test does not assume known distributions, it does not deal with parameters, and therefore it is a non-parametric test. Hence, the data need not to be normally distributed. This test essentially calculates the difference between the sets of observations and analyzes these differences to determine if they are statistically significantly different from one another. When performing the Wilcoxon Rank Sum test, we need to control the error rate across the whole study. The simplest and most conservative approach to control a study-wide error rate is Bonferroni correction, which can reduce the chances of obtaining FP results when multiple pairwise tests are performed on a single dataset. In particular, Bonferroni correction designs an adjustment to prevent data from incorrectly appearing to be statistically significant. Hence, in this study, we employ Bonferroni correction to control the error rate. In addition, the Cliff Delta Effect Size is a non-parametric effect size measure that quantifies the amount of difference between two groups of observations beyond the *p*-value interpretation.

Results. Figure 10 shows the comparison results of the three approaches, where *Our* stands for our proposed approach. We can see that our approach achieves the best and balanced results compared to the two baseline approaches. In the Java projects, we can see that our approach achieves the Precision of 93.24%. In contrast, *DeepMethod* and *RenameExpander* only achieve 73.89% and 66.63% in the same condition. This show that our approach outperforms the baseline approaches by almost 20% in terms of the Precision in the Java projects. As for the Recall, we can see that *DeepMethod* achieves the best results (i.e., 96.81%). Even though our approach achieves the Recall of 86.03%, it still outperforms *RenameExpander* by almost 30%. In terms of the F-measure, we can find that our approach can balance Precision and Recall so as to achieve the best F-measure. Specifically, our approach achieves the F-measure of 89.26%. However, *DeepMethod* and *RenameExpander* only achieve 80.96% and 61.47%, respectively.

We obtain similar results in the Android projects. For example, our approach achieves the best Precision of 92.66% and outperforms *DeepMethod* by 32.70% and *RenameExpander* by 37.82%. Even though our approach achieves the second largest Recall, which is less than *DeepMethod* by 5.74%, our approach achieves the best F-measure and significantly outperforms the baseline approaches. Specifically, our approach achieves the F-measure of nearly 90% (i.e., 89.82%). In contrast, *DeepMethod* and *RenameExpander* only achieve 68.44% and 46.48%, respectively.

We employ the *p*-values and the effect size to evaluate the statistical results. The statistic hypothesis test is usually transformed to a probability scale and expressed as a *p*-value. The *p*-values provide information about how big is a difference between the two groups of results of our approach and the baseline approaches. Meanwhile, the effect size measures the sizes of differences between group means, which is a good supplement to the *p*-values. As the results of the statistic hypothesis test, all the *p*-values of approach comparison for all the evaluation metrics are less than 0.05, which means that there are significant differences among the performance of those approaches. In addition, most of the effect sizes are large, with only two small and one medium. Considering that the average values achieved by our approach are better than those of the two baseline approaches, we can conclude that our approach achieves significantly better results than the two baseline approaches.

From these experimental results, we can find that even though our approach is based on a traditional classification technique, it is still superior to the existing deep learning based approach (i.e., *DeepMethod*). This means that deep learning approaches do not always achieve the best results. The reason our approach outperforms the existing approaches may be because we systemically investigate why developers rename identifiers (see Section 2.2), from which we

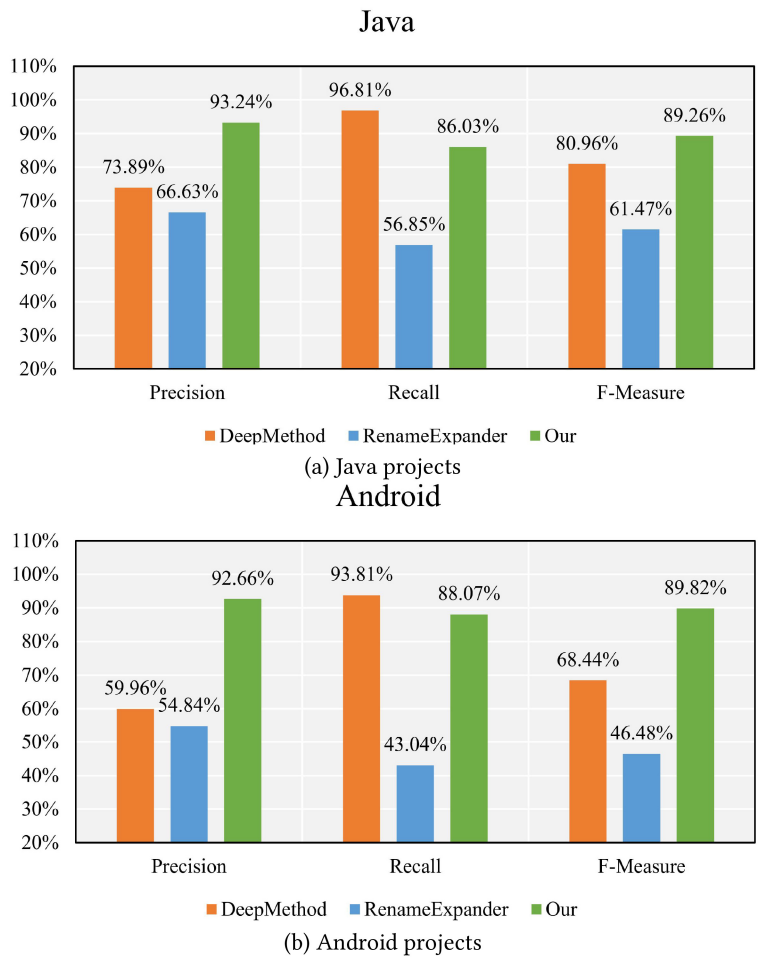


Fig. 10. The comparison results of different identifier renaming prediction approaches.

obtain the key features that could help detect those identifiers needing renaming. These features could well detect the domain specific knowledge of identifiers themselves from the lexical and semantic aspects. In contrast, deep learning based approaches find important features based on a large scale of training data from scratch, without knowing developers' actual development intents. Hence, for the future work of identifier renaming prediction, it is better to dig more about the domain-specific knowledge of identifiers. In addition, it is worthy to explore how to combine the domain specific knowledge with deep learning based approaches to further improve the performance of identifier renaming prediction.

Conclusion

Our approach can balance Precision and Recall, so it achieves the best F-Measure among the existing approaches in both the Java and Android projects. In addition, all the p-values of all the evaluation metrics are less than 0.05. Most of the effect sizes are large, with only two small and one medium. This shows that our approach achieves significantly better results than the two baseline approaches.

5.1.2 Influence of Proposed Features.

RQ2: What Is the Influence of the Proposed Groups of Features to the Prediction Results of Our Approach?

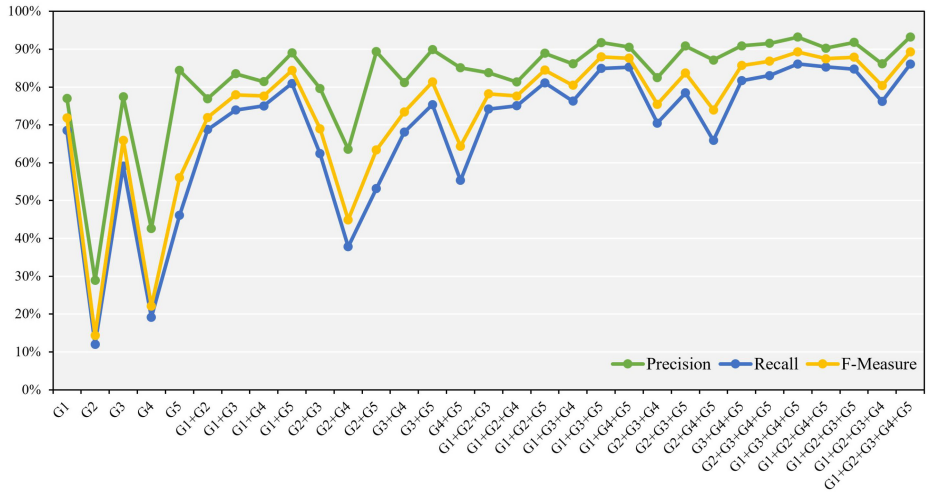
Motivation. To better characterize different granularities of identifiers from different aspects, we propose five groups of features. Different groups and combinations of groups of features may contribute differently to the prediction of identifiers that need renaming. To explore such influence of our proposed features, we set up this RQ.

Approach. We verify our approach by leveraging different groups and combinations of groups of features. Specifically, we first employ each single group of features and validate the prediction performance of our approach. Then, we test various combinations of groups of features and evaluate the results of our approach. By comparing the achieved Precision, Recall, and F-measure of our approach in the Java and Android projects, we can know the influence of groups of features to our approach.

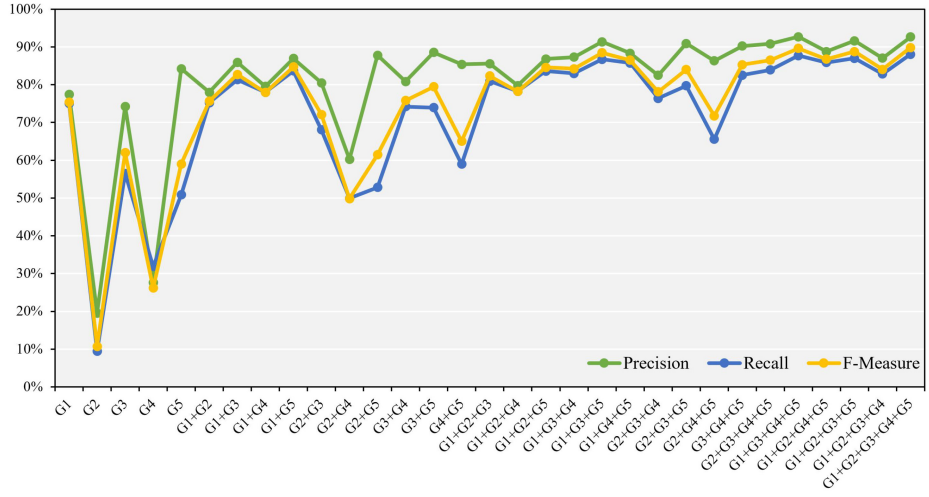
Results. Figure 11 shows the prediction results of our approach when employing different groups of features. When comparing different single group of features, we can see that employing feature group 1 (i.e., the Inherence group), our approach could achieve the best overall results. For example, our approach achieves the F-measure of 71.87% in the Java projects and 75.34% in the Android projects, which are the largest values among different single groups of features. Our approach with feature group 3 (i.e., the Relation with Code Entities group) achieves the second largest F-measure values in both the Java and Android projects. In contrast, only employing feature group 2 (i.e., the Relation with Convention), our approach achieves the worst values of F-measure. This is simply because feature group 1 and feature group 3 contain more features than feature group 2. In addition, even though feature group 5 (i.e., the History group) only includes three features, our approach only with this feature group could also achieve relatively good results. For example, our approach with feature group 5 achieves the best Precision (i.e., 84.40% and 84.23% in the Java and Android projects, respectively) but moderate Recall (i.e., 46.10% and 50.86% in the Java and Android projects, respectively) and F-measure (i.e., 56.06% in the Java projects and 59.02% in the Android projects). This means that employing the historical features is effective.

Comparing different combinations of feature groups, we can see that along with the increase in the groups of features, our approach achieves better results overall. Taking the synthetic evaluation metric F-measure as an example, when combining only two groups of features, our approach achieves the F-measure at the range between 44.89% to 84.38% in the Java projects and 49.83% to 84.71% in the Android projects. When combining three groups of features, our approach achieves the F-measure of 73.94% to 87.97% in the Java projects and 71.74% to 88.45% in the Android projects. Furthermore, when combining four groups of features, our approach obtains the range of F-measure values from 80.40% to 89.27% in the Java projects and 84.03% to 89.60% in the Android projects. We can see that along with the increase of the number of features, both the maximum and minimum values of F-measure achieved by our approach are improved. In addition, when employing all five groups of features, our approach achieves almost the best results in terms of Precision, Recall, and F-measure. For example, the achieved Precision values are more than 92% in both the Java and Android projects when using all the proposed features. The F-measure values achieved by our approach are nearly 90% in the Java and Android projects by employing all the features.

From Figure 11, we can also find that the performance of our approach does not strictly show the upward trends along with the increase of feature groups. In some situations, using more features (groups) could not certainly achieve better results. For example, some combinations of three features are not as good as combinations of just two (e.g., the performance of our approach with G3+G5 outperforms that of our approach with G2+G3+G4). After digging more about the features, we find that feature group 2 and feature group 4 do not perform as well as the other three groups,



(a) The results in the Java projects



(b) The results in the Android projects

Fig. 11. The influence of different groups of features to the prediction performance of our approach.

leading the combinations of feature groups involving 2 and 4 do not perform well accordingly. There are two potential reasons that feature groups 2 and 4 do not perform well. First, feature groups 2 and 4 contain four and five features, respectively, which are less than the number of features in group 1 containing 10 features and group 3 containing 7 features. The small number of features limits the performance of feature groups 2 and 4. Second, feature group 2 checks the relationship between identifiers and their corresponding code conventions (e.g., style and POS), whereas feature group 4 detects the characteristics of the enclosing files of identifiers. Both of the two features groups detect the lexical properties of identifiers, which may not perform as well as those feature groups measuring the semantic aspects of identifiers (e.g., feature group 5). Hence, even though feature group 5 contains only 3 features, it measures the semantic change histories of identifiers so as to achieve better performance than feature groups 2 and 4.

When comparing the Java projects against the Android projects, we can see from Figure 11 that the influence of different groups of features is consistent in the Java and Android projects.

This means that when our approach with some features achieves good (or bad) results in the Java projects, our approach with the same features also achieves good (or bad) results in the Android projects. For example, when considering the combination of two groups of features, our approach with the feature group 1 and feature group 5 achieves the best F-measure of 84.38% in the Java projects. It also achieves the best F-measure of 84.71% in the Android projects with the same feature groups. In contrast, our approach with feature group 2 and feature group 4 achieves the worst F-measure (i.e., 44.89% in the Java projects and 49.83% in the Android projects). For the other groups and group combinations of features, our approach also achieves consistent results. Hence, these proposed features are with fine robustness.

Conclusion

Different groups of features contribute differently to the prediction performance of our approach. Along with the increase of the number (groups) of features, our approach achieves increasingly better results. Feature group 2 and feature group 4 do not perform as well as the other three groups, since the two feature groups contain a small number of features and measure the lexical aspects of identifiers. In addition, different groups of features achieve consistent results in both the Java and Android projects.

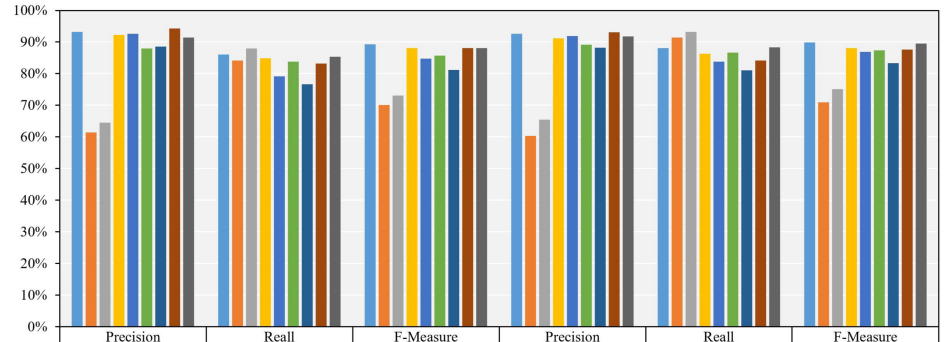
5.1.3 Effectiveness of the Default Classifier.

RQ3: Is the Default Random Forest Classifier Effective in Predicting Identifiers That Need Renaming?

Motivation. When we design our approach, we employ the widely used Random Forest classifier by default. In addition, we do not adjust the values of parameters and keep them as the default values in WEKA for the Random Forest classifier. The classifier is an important component in our approach, which could heavily impact the performance of our approach in predicting those identifiers that need renaming. If we choose an inappropriate classifier, it may have a negative impact on the prediction performance of our approach. On the contrary, if we select an appropriate one, it may greatly improve the prediction performance of our approach. To validate whether the default Random Forest classifier is effective in our approach, we set up this RQ.

Approach. To show the effectiveness of the default Random Forest classifier, we compare it with the other eight commonly used classifiers, including Naive Bayes, Bayes Network, Decision Tree, Simple Logistic, KNN, AdaBoost, Bagging, and MLP. These classifiers belong to different categories and have a broad cover range for different tasks. For example, Naive Bayes and Bayes Network are Bayes-based classifiers. Decision Tree is a tree-based classifier. Simple Logistic and MLP are function-based classifiers. Specifically, in each run, we only employ one of these comparison classifiers to replace the default Random Forest classifier and keep the other components in our approach the same. We implement all the classifiers based on WEKA. These comparison classifiers also have their own parameters. The same as the default Random Forest classifier, we set the parameters as their default values in WEKA for the comparison classifiers, for two reasons. First, we can make a fair comparison and better evaluation to the effectiveness of each classifier. Second, existing studies have shown that the default values of parameters in WEKA have been well calibrated so that they can help achieve good results. In such a way, by comparing the results of different classifiers, we can know which classifier can help our approach achieve the best results. In addition, we conduct the Wilcoxon Rank Sum test to figure out whether there are statistically significant differences between the Random Forest classifier and other comparison classifiers.

Results. Figure 12 presents the results of our approach with different classifiers on the Java and Android projects. We can see that our approach with the default Random Forest classifier achieves



	Java			Android		
	Precision	Reall	F-Measure	Precision	Reall	F-Measure
Random Forest	93.24%	86.03%	89.26%	92.66%	88.07%	89.82%
Naïve Bayes	61.45%	84.17%	70.10%	60.30%	91.45%	70.87%
Bayes Net	64.47%	87.93%	73.08%	65.50%	93.17%	75.09%
Decision Tree	92.21%	84.85%	88.10%	91.13%	86.26%	88.04%
Simple Logistic	92.66%	79.14%	84.73%	91.95%	83.75%	86.91%
KNN	87.95%	83.84%	85.65%	89.11%	86.60%	87.35%
AdaBoost	88.55%	76.62%	81.23%	88.16%	81.05%	83.30%
Bagging	94.33%	83.25%	88.03%	93.13%	84.11%	87.64%
Multilayer Perceptron	91.46%	85.32%	88.03%	91.78%	88.33%	89.48%

Fig. 12. The results of our proposed approach with different classifiers.

the best results overall. When comparing the Random Forest classifier with the Bayes-based classifiers (i.e., the Naive Bayes and Bayes Network classifiers), we can see that our approach with the Naive Bayes and Bayes Network classifiers achieves much lower Precision values than our approach with the Random Forest classifier. Even though our approach with the Bayes Network classifier achieves slightly better Recall values than our approach with Random Forest, our approach with Random Forest achieves much better F-measure values. When comparing the Random Forest classifier with the rest of the comparison classifiers, we can find that our approach with Random Forest achieves the best Precision, Recall, and F-measure values in almost all situations. Even though the rest of the comparison classifiers can help our approach achieve relatively good Precision values in some situations, the corresponding Recall values are not satisfying, leading to lower F-measure values. For example, when comparing Random Forest with Bagging, our approach with the Random Forest classifier achieves the Precision of 93.24%, Recall of 86.03%, and F-measure of 89.26% in the Java projects. In contrast, our approach with the Bagging classifier achieves the Precision of 94.33%, Recall of 83.25%, and F-measure of 88.03% in the same situation.

When comparing the Java projects with the Android projects, we can observe that our approach with the same classifier achieves consistent results. Taking the Decision Tree classifier as an example, our approach with this classifier achieves the Precision of 92.21% in the Java projects. Similarly, our approach with the same classifier achieves the Precision of 91.13% in the Android projects. For the Recall values, our approach with the Decision Tree classifier achieves 84.85% and 86.26% in the Java and Android projects, respectively. In addition, in terms of F-measure, our approach could achieve 88.10% and 88.04% in the Java and Android projects, respectively. Hence, our approach with the same classifier achieves similar results in both the Java and Android projects.

As for the results of the statistical hypothesis test, taking the F-Measure as an example, the Random Forest classifier has a statistically significant difference with the Naive Bayes classifier, the Bayes Net classifier, the Simple Logistic classifier, the KNN classifier, and the AdaBoost classifier in both the Java and Android projects, where the p-values are less than 0.05. This means that the Random Forest classifier achieves significantly better results than these classifiers, considering that the average F-Measure achieved by the Random Forest classifier is also larger than the average

F-Measure achieved by these classifiers. In contrast, the Random Forest classifier does not have a statistically significant difference with the Decision Tree classifier, the Bagging classifier, and the MLP classifier in both the Java and Android projects, where the p-values are larger than 0.05. In addition, the Random Forest classifier achieves a similar average F-Measure as these classifiers in the experimental projects. Hence, the Random Forest classifier achieves results similar to these classifiers. This shows that our approach can also employ these classifiers as the default classifier.

For the future identifier renaming prediction approaches, it is better to conduct some preliminary experiments to explore the influence of employing different classifiers, if it is regarded as a classification problem. In such a way, we can figure out the exact classifier that could achieve the best performance. In addition, it is interesting to explore how to combine different classifiers to form an ensemble classifier. In such a way, the ensemble classifier corrects the errors produced by a single classifier so as to achieve better results.

Conclusion

The default Random Forest classifier is effective in our approach. Our approach with some classifiers (e.g., the MLP classifier) can also achieve results similar to our approach with the default Random Forest classifier. When comparing the Java projects with the Android projects, our approach with a same classifier achieves consistent results.

5.1.4 Performance in Cross-Project Prediction.

RQ4: What Is the Performance of Our Approach in the Context of Cross-Project Prediction?

Motivation. The prediction ability of our approach depends not only on the employed algorithm but also the quality of the training data. Existing studies usually evaluate the proposed approach in the within-project context, which assumes the existence of previous sufficient data for a specific project. However, there are a lot of projects in practice that do not have sufficient historical data, especially for new projects. In such a situation, an alternative approach is to form a training set that is composed of external projects, which is called *cross-project prediction*. Cross-project prediction is a good complement to within-project prediction. To better evaluate our approach in different contexts and validate whether our approach still outperforms the baseline approaches in the context of cross-project prediction, we set up this RQ.

Approach. There are 50 Java projects and 50 Android projects in the dataset. For each category of projects, when we evaluate one specific project, we employ the remaining 49 projects as the training projects to train the classifier. The trained classifier is validated on the specific project. When all the projects are selected as the test set once, we calculate the average result as the final result to evaluate the performance of our approach in the context of cross-project prediction. For the baseline approaches *DeepMethod* and *RenameExpander*, only *DeepMethod* can be evaluated in the context of cross-project prediction. This is because the rationale of *RenameExpander* is based on the conducted similar renaming activities in the same project to predict those identifiers that need renaming. A conducted renaming activity in a specific project cannot have any relationship with the identifiers in other projects. Hence, in this RQ, we only compare our approach against *DeepMethod* in the context of cross-project prediction. We evaluate *DeepMethod* in the same way so as to make a fair comparison.

Results. Figure 13 shows the results of our approach and *DeepMethod* in the context of cross-project prediction. We can see that even in the context of cross-project prediction, our approach could also achieve satisfactory results. For example, our approach in the context of cross-project

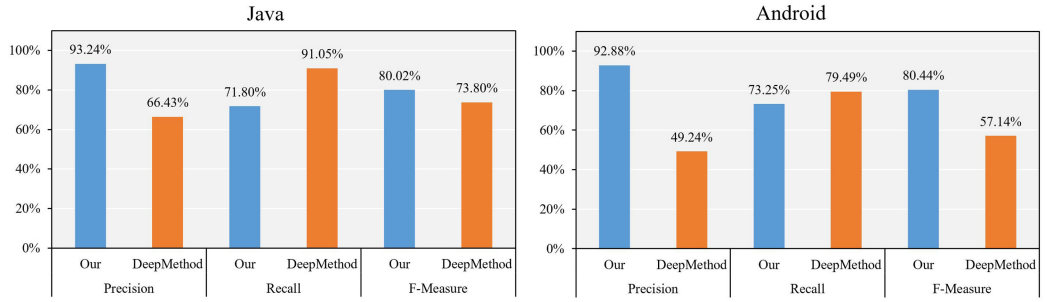


Fig. 13. The results of our approach and *DeepMethod* in the context of cross-project prediction.

prediction achieves the same Precision (i.e., 93.24%) as in the context of within-project prediction (i.e., 93.24%) in the Java projects. Even though the performance of our approach in cross-project prediction (i.e., 71.80%) is not as good as in within-project prediction (i.e., 86.03%) in terms of Recall, the achieved F-measure value can reach 80.02% in the Java projects. In addition, we can observe similar results in the Java and Android projects in cross-project prediction. For example, our approach achieves the Precision of 92.88%, Recall of 73.25%, and F-measure of 80.44% in the Android projects. This means that our approach could also achieve solid results in the context of cross-project prediction.

When comparing our approach against *DeepMethod*, we can see that our approach achieves better Precision than *DeepMethod*. For example, our approach achieves the Precision of 93.24% and 92.88% in the Java and Android projects, respectively. In contrast, *DeepMethod* only achieves 66.43% and 49.24% in the same conditions. However, our approach does not outperform *DeepMethod* in terms of Recall in the context of cross-project prediction. For instance, our approach achieves the Recall of 71.80% in the Java projects. In contrast, *DeepMethod* can achieve 91.05% in the same situation. As for the important F-Measure, we can see that our approach still outperforms *DeepMethod*. For example, our approach and *DeepMethod* achieve the F-Measure of 80.02% and 73.80% in the Java projects, respectively, where the disparity is 6.22%. Hence, our approach outperforms *DeepMethod* in the context of cross-project prediction in terms of Precision and F-Measure.

Conclusion

Even in the context of cross-project prediction, our approach could also achieve satisfactory and solid performance. In addition, compared against the baseline approach, our approach still considerably outperforms it in the context of cross-project prediction in terms of Precision and F-Measure.

5.2 Evaluation of Renaming Suggestion

We investigate the following two RQs to evaluate the performance of our approach in suggesting new identifiers. In addition, we discuss the usage scenario of our approach by providing several cases.

5.2.1 *Suggestion Performance Comparison.*

RQ5: Does Our Approach Achieve Better Results Than *RefactorLearning in Suggesting Accurate Full Name Identifiers?*

Motivation. There is an existing state-of-the-art approach for suggesting new full name identifiers, namely *RefactorLearning*. In this RQ, we aim to compare our approach against

RefactorLearning to figure out which approach could accurately suggest or recommend new identifiers for those that need renaming.

Approach. Similarly, we re-implement *RefactorLearning* by ourselves according to its main procedures. After that, we validate and compare our approach against *RefactorLearning* in both the Java and Android projects. Specifically, for each identifier that needs renaming, we run both our approach and *RefactorLearning* to obtain a recommendation list, respectively. Then, we employ Hit@K (where K ranges from 1 to 10) to measure the performance of the new identifier recommendation list. If a specific identifier recommended by our approach or *RefactorLearning* is exactly the same as the target new identifier, we call it *Hit*. By calculating the average result, we can obtain the value of Hit@K for both our approach and *RefactorLearning*. Based on the calculation process of Hit@K, the higher the value of Hit@K, the better the performance. In such a way, we can know which approach is better in suggesting new full name identifiers. In addition, we employ the Wilcoxon Rank Sum test with Bonferroni correction and Cliff Delta Effect Size to conduct the statistical hypothesis test to explore whether there are significant differences between the two approaches. Still, we employ the *p*-values and the effect size to measure the differences of the performance of our approach and *RefactorLearning*.

Results. Figure 14 shows the comparison results of our approach and *RefactorLearning*. To better present the results, we show the results from top 1 to top 10 in terms of Hit. We can see from Figure 14 that our approach obviously achieves better results than *RefactorLearning* in both the Java and Android projects. Specifically, our approach achieves Hit@1 of 21.75% and 21.00% in the Java and Android projects, respectively. In contrast, *RefactorLearning* only achieves 13.00% and 16.88% in the same condition. Along with the increase in the number of the recommended identifiers, the disparity of our approach and *RefactorLearning* becomes larger. For example, when recommending five new identifiers, our approach achieves Hit@5 of 44.07% in the Java projects, whereas *RefactorLearning* only achieves 16.04% in the same condition, where the disparity is 28.03%. When the number of recommended new identifiers increases to 10, our approach achieves Hit@10 of 48.58% and 40.97% in the Java and Android projects. In contrast, *RefactorLearning* only achieves 18.96% and 25.22% in the same condition, where the disparities are 29.62% and 15.75%.

As for the results of the statistical hypothesis test, all the *p*-values from top 1 to top 10 are less than 0.05. In addition, except for only one effect size being medium, the other effect sizes are all large. This means that there are significant differences between the performance of our approach and *RefactorLearning*. Combined with the average values shown in Figure 14, we can conclude that our approach is significantly better than *RefactorLearning* in suggesting correct new identifiers.

The reason our approach could achieve better results than *RefactorLearning* may be that our approach mines the revision patterns from the training set, which could help us obtain a series of identifier revision patterns so as to accurately generate correct identifiers. In addition, we find that developers rename identifiers following a series of renaming sequences. Thus, we also employ the identifier renaming history to help us generate correct new identifiers. By combining the two aspects, our approach can accurately suggest full name identifiers for developers.

Conclusion

Our approach achieves a better Hit Ratio than *RefactorLearning* in suggesting new full name identifiers from top 1 to top 10. All the *p*-values from top 1 to top 10 are less than 0.05. In addition, except for only one effect size being medium, the other effect sizes are all large. This means that there are significant differences between the performance of our approach and *RefactorLearning*.

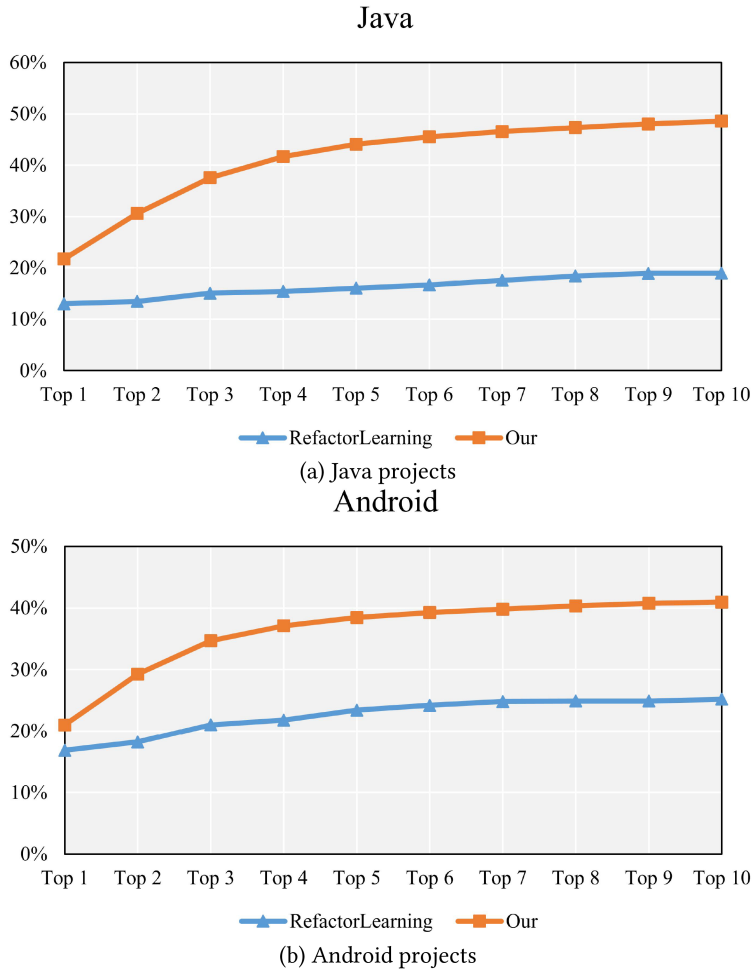


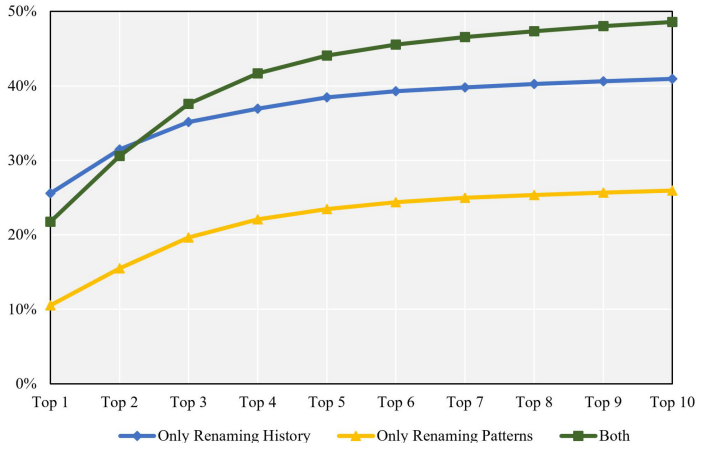
Fig. 14. The comparison results of different identifier renaming suggestion approaches.

5.2.2 Effectiveness of Change Patterns and History.

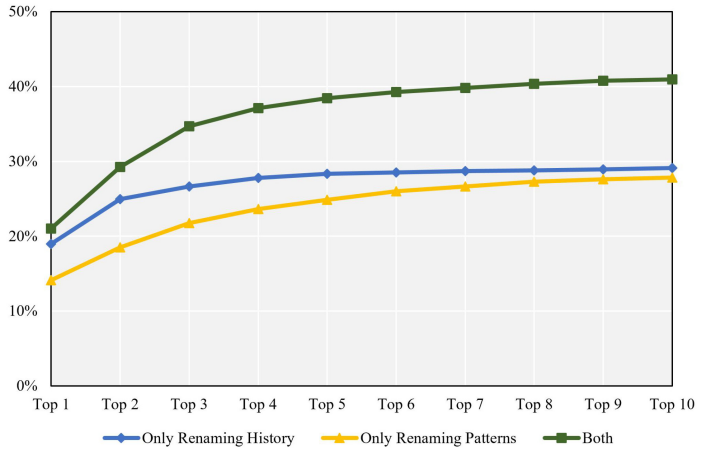
RQ6: Does Combining Both the Information of Renaming History *and Renaming Patterns Could Help Improve the Performance of Our Approach in Suggesting New Identifiers?*

Motivation. When we design our new identifier suggestion approach, we fully leverage the renaming history and the renaming patterns of related code entities, since we find that developers tend to follow some renaming patterns based on a series of renaming sequences shown in Section 2.3. The renaming history and the renaming patterns may contribute differently to the performance of our approach. To explore their contributions and whether combining them could improve the performance of our approach in suggesting new identifiers, we set up this RQ.

Approach. We define two variants of our approach. The first variant only uses the renaming history to calculate relevant scores of the generated new identifiers and regards the relevant scores as final scores. In contrast, the second variant of our approach only employs the renaming patterns to compute the final relevant scores of the newly generated identifiers. By comparing our original approach with its two variants, we can know whether combining the two types of information (i.e., the renaming history and the renaming patterns) could improve the performance of our approach.



(a) The results in the Java projects



(b) The results in the Android projects

Fig. 15. The experimental results of our approach against its variants.

Results. Figure 15 presents the experimental results of our approach against its two variants. Similarly, we present the Hit@K achieved by different approaches from top 1 to top 10 in Figure 15, where *Both* means our original approach, and *Only Renaming History* and *Only Renaming Patterns* are the two variants of our approach only using the renaming history and only using the renaming patterns, respectively. When we focus on the two variants, we can see that the trend line of our approach only employing the renaming history is higher than that of our approach with only the renaming patterns. This shows that our approach with only the renaming history achieves better results than our approach only with the renaming patterns. For example, our approach with only the renaming history achieves Hit@5 of 38.45% and 28.34% in the Java projects and the Android projects. In contrast, our approach with only the renaming patterns achieves 23.47% and 24.87% in the same condition. When K increases to 10, our approach with only the renaming history achieves Hit@10 of 40.93% and 29.11% in the Java and Android projects. However, our approach with only the renaming patterns achieves 25.92% and 27.82% in the same situation. Hence, we can obtain that employing the renaming history is more effective than employing the renaming patterns.

Fig. 16. Cases of suggested identifiers of our approach.

When comparing our approach as a whole with its two variants, we can see that our original approach achieves better results than its variants overall. We can see that almost all the results achieved by our approach are better than those achieved by its variants, with only two exceptions of top 1 and top 2 in the Java projects. Even though Hit@1 and Hit@2 achieved by our approach are slightly lower than its variant with only the renaming history, these results are still larger than its variant with only the renaming patterns. For the other situations, our approach achieves the best results. For example, our original approach achieves Hit@10 of 48.58% and 40.97% in the Java and Android projects. In contrast, its variants with the renaming history achieve 40.93% and 29.11%, and its variants with the renaming patterns achieve 25.92% and 27.82% in the same situation. This means that combining both the renaming history and the renaming patterns could help improve the performance of our approach in suggesting new identifiers.

Conclusion

Our approach only with the renaming history achieves better results than our approach only with the renaming patterns. When merging the two aspects, our approach achieves better results than its two variants. Combining both the renaming history and the renaming patterns could improve the performance of our approach in suggesting new identifiers.

5.2.3 Usage Scenario. To better present the advantages and disadvantages of our new identifier suggestion approach, we discuss the usage scenario of our approach. Specifically, we select several cases from the results of our approach, in which three cases are from the Java projects and three cases come from the Android projects. Among the three cases in both the Java and Android projects, two cases are successful cases, which means that our approach successfully suggests the target

identifiers in the top 10 ranked list. One case is the failed case showing that our approach could not generate the exactly correct target identifier in the top 10 ranked list.

Figure 16 presents the cases of the suggested identifiers by our approach. In each case, we display three parts to better show the suggestion results, including the flawed identifier (in the yellow background), the corresponding target correct identifier (in the green background), and the suggested identifiers by our approach (the correctly suggested one is also shown in the green background). In terms of the first case from the Java projects, we can see that the flawed identifier *lock* shall be renamed to the target identifier *refreshLock*, in which a specific token *refresh* is added to make it concrete. Our approach suggests a list of new identifiers, where the top 1 ranked identifier is exactly the same as the target identifier. The reason is that a generated revision pattern shows that the token *refresh* is often added in related code entities to narrow down their meanings. Hence, our approach could recognize this common addition renaming pattern to correctly generate the target identifier. In such a way, developers can easily rename this flawed identifier with the assistance of our approach. As for the second case in the Java projects, the flawed identifier *convertToString* needs to be renamed as *convertIntArrayToString*, where specific tokens *Int* and *Array* should be added in the middle. By analyzing the renaming history and the renaming patterns of related code entities, our approach generates a ranked new identifier list, in which the top 5 ranked identifier hits the target identifier. In such a situation, developers could also take little time to find the correct identifier in the suggested identifier list. By analyzing the reason, we also find that the specific tokens *Int* and *Array* are added to similar identifiers. Hence, this identifier shall be also added with the two tokens. In contrast, in the third case, the flawed identifier *parseImageSize* is expected to be renamed to the target identifier *parseGIFImageSize*, where the format of the image *GIF* shall be added to make it specific. Even though our approach fails to suggest the exact correct identifier in the new identifier list, we believe that developers can still benefit from this suggested identifier list. Developers can easily think out the target identifier *parseGIFImageSize* from the top 1 suggested identifier *parsePNGImageSize* and the top 2 suggested identifier *parseJPEGImageSize*, since these identifiers follow the same patterns but only with a different image format.

As for the cases from the Android projects, we can see that the flawed identifier *feedbackTitle* should be renamed to *feedbackType* in the fourth case, in which the token *Title* is changed to *Type*. Our approach detects such a renaming pattern so that it can suggest the correct target identifier in the top 1 ranked list. In terms of the fifth case, the flawed identifier *square_btn* shall be renamed to *sequare_white*. Our approach finds that many related code entities are renamed into this target identifier, so it can successfully suggest the target identifier in the top 4 ranked identifier. In the last case, the flawed identifier *dgDrawableHeight* is renamed to *dgDrawableMargin*. Our approach fails to recommend the target identifier. However, we can find that the identifier *dg_drawable_margin* is recommended, which is the same as the target identifier except for the identifier style. Developers may also draw out the correct target identifier by modifying this identifier with few manual operations. Hence, developers can also benefit from this identifier suggestion list by our approach.

From the successful and failed cases of our approach, we find that our approach performs well in the usage scenario where there are obvious renaming patterns from similar identifiers. These renaming patterns can be detected and further applied in the generation process of new identifiers. However, in the usage scenario where there is little obvious renaming pattern of similar identifiers or there is little renaming history in the software projects, our approach can hardly leverage the renaming patterns and the renaming history. In such a situation, even though our approach could still recommend some new identifiers for those that need renaming, it may fail to recommend the correct new identifiers.

Conclusion

Even though our approach fails to suggest correct target identifiers in some situations, developers can still benefit from the suggested identifier list.

6 THREATS TO VALIDITY

The experimental setup could inevitably influence the experimental results. In this section, we discuss the threats to validity of our study, including threats to internal validity, threats to conclusion validity, and threats to external validity.

6.1 Threats to Internal Validity

When we design our identifier renaming prediction approach, we regard it as a classification problem and employ the Random Forest classifier to resolve it by default. The values of parameters in the Random Forest classifier can influence the results of our approach, which may be an internal validity. When we implement the Random Forest classifier, we keep the parameters as their default values in WEKA without tuning them. Evidently, if we elaborately adjust the values of these parameters, we can obtain better results in predicting identifiers that need renaming. There are two reasons. First, we want to make a fair evaluation of our approach to show its potential so that it can be easily configured and generalized to other datasets. Second, existing studies have shown that the default values of parameters in WEKA could also achieve good results. Hence, we argue that the performance of our approach with the default parameters is justifiable. In the future, we will explore the impacts of parameters in classifiers to the results of our approach.

When we design our identifier suggestion approach, we fully consider the information from two aspects: the renaming history and the renaming patterns. The two types of information are regarded as equal to the calculation of the final scores of newly generated identifiers. Different weights in the renaming history and the renaming patterns could lead to different results in our approach, which may be a potential internal threat. Assigning the same weight to both the renaming history and the renaming patterns is the simplest way to obtain the final results, which could make our approach easy to configured and adapt to different situations. We will investigate such impacts as future work.

In this study, we construct the ground truth dataset from the experimental Java and Android projects. The construction process of the ground truth dataset may be associated with some internal threats. For example, when we trace back the experimental projects into historical versions, we set the time interval as no less than 6 months, which may be a potential threat. The development speed and process of different projects are different. Some projects may update and evolve quickly (e.g., less than 6 months). In contrast, the other projects may update a new version slowly (e.g., more than 6 months). Hence, it is hard to decide a reasonable time interval. In the future, we will explore the other time intervals to investigate whether the time to the historical version can influence the performance of our approach.

The last internal threat is that the implemented baseline approaches may not faithfully represent the original approaches described in their papers. There may be potential bugs in our re-implementation. To reduce such a threat, we carefully check the details of these baseline approaches. In addition, we use the same values of hyper-parameters as the original papers, if there are any. The authors also employ the code quality assurance mechanisms for the re-implementation. For example, we employ the code review to carefully check the experimental

scripts to ensure their correctness. As well, we obtain confirmation from the original authors if there are any questions and ambiguities.

6.2 Threats to Conclusion Validity

Threats to conclusion validity deal with the relationship between the data treatment and the results. The statistical conclusion validity is concerned with the validity of the statistical analysis method used in this study. We appropriately used the Wilcoxon Rank Sum test with Bonferroni correction and Cliff Delta Effect Size to explore the statistically significant differences of the obtained results for some RQs. The Wilcoxon Rank Sum test is a typical and robust non-parametric test, which does not make any assumption on the data distribution. In addition, Bonferroni correction is used to correct and adjust the *p*-values. Hence, it is not likely that the results of the statistical test contain errors. Moreover, the observed differences are highlighted by different values of the effect size. Hence, we believe that the threats to conclusion validity are considered under control.

6.3 Threats to External Validity

In this study, we employ the widely used publicly available dataset published by Allamanis and Sutton [5] and randomly select 100 Java and Android projects to conduct the experiments. The experimental results on the selected projects may be not generalized to other datasets. However, these projects are randomly sampled, which could present the real scenarios to a great extent, since these projects belong to different project families with different popularity and sizes. In the future, we will extend our approach by validating it with more projects.

The second possible external validity may be the identifier renaming prediction and suggestion context. In this study, we validate the effectiveness of our identifier renaming prediction approach in both within-project and cross-project contexts. In contrast, we only validate the effectiveness of our identifier renaming suggestion approach in the within-project context. This is because for the identifier renaming prediction task, different projects may follow similar or the same renaming patterns or reasons (e.g., conflict with code conventions and other code entities). In such a way, we can learn the common renaming patterns across different projects. However, for the identifier renaming suggestion task, we need to fully analyze the current programming context to generate the correct identifiers. It is impossible to generate correct identifiers based on the programming context of other projects. Other related studies also validate their identifier renaming suggestion approaches only in the within-project context [41]. Hence, we also validate our identifier renaming suggestion approach in the within-project context.

7 RELATED WORK

In this section, we present the closely related studies about identifier renaming. Typically, the identifier renaming process usually involves three main steps, including identifier renaming pre-processing, identifier renaming prediction and suggestion, and identifier renaming execution [34]. Hence, we illustrated related studies based on the three steps. Next, we introduce the typical studies of the related work. In addition, we discuss the differences between our study and existing studies, especially how our study advances existing ones.

7.1 Identifier Renaming Pre-processing

Identifier renaming pre-processing aims to normalize identifiers into several natural language tokens so that developers can easily understand the lexical and semantic meaning of identifiers when they inspect whether specific identifiers need renaming. Typically, identifier renaming pre-processing contains two primary steps: identifier splitting and identifier expansion. The two steps can be investigated either independently or simultaneously.

Identifier splitting splits a specific identifier into a set of tokens. Enslen et al. [21] proposed *Samurai*, an approach for identifier splitting by mining token frequencies in source code. *Samurai* split identifiers by character cases and digits for mix-case identifiers and by global and local token frequencies for same-case identifiers. Guerrouj et al. [26] proposed *TRIS*, a tree-based identifier splitter. *TRIS* transformed dictionary tokens into a weighted tree presentation and further searched the optimal split (the shortest path) in the weighted tree. Corazza et al. [18] proposed *LINSEN*, an efficient identifier splitting approach. *LINSEN* was based on a pattern matching algorithm (i.e., the Baeza-Yates and Perleberg algorithm). Guerrouj et al. [25] further proposed *TIDIER*. *TIDIER* was based on an adaptation of the dynamic time warping algorithm to identify tokens in continuous speech, which could be used when code conventions (e.g., camel case) were not used and abbreviations were contained. Recently, deep neural networks were employed to resolve the research task of identifier splitting [35, 42]. These approaches usually achieved the best results, which an Accuracy of more than 90% on average.

After identifiers are successfully split into several tokens, some of them can be abbreviations. Hence, these abbreviations need to be further expanded into their correct expansions. Actually, some of the approaches mentioned previously could also expand identifiers, including *TRIS* [26], *LINSEN* [18], and *TIDIER* [25]. In addition to these approaches, Lawrie and Binkley [32] proposed a general-purpose identifier expander named *Normalize*. This approach searched expansions in source code, domain specific documentation, and Google co-occurrences dataset. Carvalho et al. [17] proposed *LIDS* to split and expand identifiers. *LIDS* employed general abbreviation dictionaries and a specific dictionary generated from software documentation to expand identifiers. Jiang et al. [29, 30] proposed a general and accurate approach to expand abbreviations using semantic relation and transfer expansion. This approach constructed a knowledge graph to represent the relationships of code entities and searched the knowledge graph to obtain expansions.

Identifier renaming pre-processing is a pre-step of identifier renaming prediction and suggestion. In this study, we employ INTT to process identifiers when we extract the proposed features for identifiers. In the future, we will try some recent approaches to process identifiers and explore their influences.

7.2 Identifier Renaming Prediction and Suggestion

The aim of identifier renaming prediction and suggestion is to identify those identifiers that need renaming and further provide a suggestion list for them. The two tasks can be investigated individually or jointly.

De Lucia et al. [19] proposed *COCONUT* to help developers keep identifiers and comments consistent with high-level artifacts. *COCONUT* could recommend candidate identifiers built from high-level artifacts related to current source code. Abebe and Tonella [1] proposed an identifier suggestion approach leveraging concepts and relations extracted from the source code. This approach ranked identifiers based on the current programming context and could be used to either complete the identifier being written or rename an existing identifier. Allamanis et al. [2] proposed an approach named *NATURALIZE* to learn the natural code conventions. *NATURALIZE* applied the statistical natural language processing technique to learn code conventions based on a large code repository, which could suggest natural identifier names and formatting conventions for variables, methods, and types. Raychev et al. [50] proposed a new approach for predicting program properties (e.g., names of identifiers) from massive codebases. Allamanis et al. [3] proposed a method and class name suggestion approach based on a neural probabilistic language model. Kim and Kim [31] proposed an approach to detect inconsistent identifiers (as a renaming opportunity) based on a special code dictionary constructed from popular API documentation as well as dominant POS. Lin et al. [38] evaluated the meaningfulness of the identifier recommendations generated by

three techniques and further proposed their own approach named *LEAR* for local variables and parameters.

Except for the preceding typical studies for identifier renaming prediction and suggestion, there are two state-of-the-art identifier renaming prediction approaches (i.e., *DeepMethod* and *RenameExpander*) and one state-of-the-art identifier renaming suggestion approach (i.e., *RefactorLearning*) as mentioned in Section 4.3. Our approach advances the state-of-the-art baseline approaches, since our approach is a lightweight approach based on the general rationale. First, compared against *DeepMethod* and *RefactorLearning*, which are deep learning based approaches, our approach is a lightweight approach. To achieve better results, both *DeepMethod* and *RefactorLearning* need to train a lot of parameters (e.g., weights in the network) based on a large scale of training data. In such a way, these approaches may consume a lot of computing resources. In contrast, there are only several parameters (e.g., the parameters in the Random Forest classifier) in our approach, which are easy to adjust and train. Second, the rationale of *RenameExpander* is only based on the conducted renaming activities. This means that this approach relies on historical renaming data to predict those related identifiers that need renaming. Once there are not sufficient historical renaming data, this approach may not work well. In contrast, our approach is based on a more general rationale, which considers not only the historical renaming data but also the lexical characteristics of identifiers and relationships between identifiers and other code entities. Hence, compared against existing state-of-the-art approaches, our approach is lightweight based on the general rationale, making our approach easy to use.

7.3 Identifier Renaming Execution

After flawed identifiers are detected and new identifiers are suggested, the last key step of automatic identifier renaming is renaming execution—that is, propagate the renamed identifiers to other parts of source code. Executing identifier renaming is also challenging, since developers should keep the code consistent and runnable with the same behaviors. Some researchers focused on proposing renaming execution approach within a single programming language. For example, Feldthaus et al. [23] proposed an approach for specifying and implementing refactorings (including identifier renaming) based on pointer analysis targeted toward JavaScript. Ge et al. [24] proposed *BeneFactor* to extract a set of manual refactoring workflow patterns. Based on these refactoring workflow patterns, *BeneFactor* detected developers' manual refactoring and helped them complete their refactoring activities automatically. Overbey et al. [46] proposed a new approach to check for behavior preservation when refactoring code (e.g., renaming identifiers) based on a differential precondition checker. Other researchers tried to propose renaming and refactoring approach for multiple programming languages. For example, Schink et al. [55] renamed identifiers by adding annotations to object relational mappings toward the Hibernate framework. Mayer and Schroeder [43] renamed multiple language applications based on the binding logic for each framework.

8 CONCLUSION AND FUTURE WORK

Identifiers play an important role in source code analysis and comprehension, and flawed identifiers could lead to serious software development problems. Hence, identifiers need to be continuously renamed. In this article, we proposed a novel identifier renaming prediction and suggestion approach. In the identifier renaming prediction phase, we designed and extracted five groups of features to capture the lexical and semantic aspects of identifiers. In the identifier renaming suggestion phase, we fully leveraged the renaming history and the renaming patterns to generate a set of new identifiers. Experimental results over 100 Java and Android projects showed that our approach could identify identifiers that need renaming with an average F-measure of almost 90% and outperformed the state-of-the-art approach in the Java and Android projects. In addition, our

approach achieved better results than the state-of-the-art approach in suggesting full name new identifiers by up to 29.62% in terms of Hit@10.

In the future, we plan to extend our approach in the following directions. First, we want to validate our approach in more projects hosted in other platforms, such as SourceForge and BitBucket. Second, we plan to calibrate the parameters in our approach and employ the advanced sampling method to handle the class imbalance problem to further improve its performance. Third, we plan to develop an automatic tool or plugin encapsulating our approach to better assist developers in their daily development activities.

REFERENCES

[1] Surafel Lemma Abebe and Paolo Tonella. 2013. Automated identifier completion and replacement. In *Proceedings of the 17th European Conference on Software Maintenance and Reengineering (CSMR'13)*. 263–272.

[2] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. 2014. Learning natural coding conventions. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'14)*. 281–293.

[3] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. 2015. Suggesting accurate method and class names. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering (FSE'15)*. 38–49.

[4] Miltiadis Allamanis, Earl T. Barr, Premkumar Devanbu, and Charles Sutton. 2018. A survey of machine learning for big code and naturalness. *ACM Computing Survey* 51, 4 (2018), Article 81, 37 pages.

[5] Miltiadis Allamanis and Charles Sutton. 2013. Mining source code repositories at massive scale using language modeling. In *Proceedings of the 10th Working Conference on Mining Software Repositories (MSR'13)*. 207–216.

[6] Maurício Aniche, Erick Maziero, Rafael Durelli, and Vinicius H. S. Durelli. 2022. The effectiveness of supervised machine learning algorithms in predicting software refactoring. *IEEE Transactions on Software Engineering* 48, 4 (2022), 1432–1450.

[7] Venera Arnaoudova, L. M. Eshkevari, M. D. Penta, Rocco Oliveto, Giuliano Antoniol, and Y. G. Gueheneuc. 2014. REPENT: Analyzing the nature of identifier renamings. *IEEE Transactions on Software Engineering* 40, 5 (2014), 502–532.

[8] Lingfeng Bao, Xin Xia, David Lo, and Gail C. Murphy. 2021. A large scale study of long-time contributor prediction for GitHub projects. *IEEE Transactions on Software Engineering* 47, 6 (2021), 1277–1298.

[9] Gabriele Bavota, Rocco Oliveto, Malcom Gethers, Denys Poshyvanyk, and Andrea De Lucia. 2014. Methodbook: Recommending move method refactorings via relational topic models. *IEEE Transactions on Software Engineering* 40, 7 (2014), 671–694.

[10] Dave Binkley, Marcia Davis, Dawn Lawrie, Jonathan I. Maletic, Christopher Morrell, and Bonita Sharif. 2013. The impact of identifier style on effort and comprehension. *Empirical Software Engineering* 18, 2 (2013), 219–276.

[11] Dave Binkley, Matthew Hearn, and Dawn Lawrie. 2011. Improving identifier informativeness using part of speech information. In *Proceedings of the 8th Working Conference on Mining Software Repositories (MSR'11)*. 203–206.

[12] Simon Butler. 2012. Mining Java class identifier naming conventions. In *Proceedings of the 34th International Conference on Software Engineering (ICSE'12)*. 1641–1643.

[13] Simon Butler, Michel Wermelinger, Yijun Yu, and Helen Sharp. 2010. Exploring the influence of identifier names on code quality: An empirical study. In *Proceedings of the 14th European Conference on Software Maintenance and Reengineering (CSMR'10)*. 156–165.

[14] Simon Butler, Michel Wermelinger, Yijun Yu, and Helen Sharp. 2011. Mining Java class naming conventions. In *Proceedings of the 27th International Conference on Software Maintenance (ICSM'11)*. 93–102.

[15] Jürgen Börstler and Barbara Paech. 2016. The role of method chains and comments in software readability and comprehension—An experiment. *IEEE Transactions on Software Engineering* 42, 9 (2016), 886–898.

[16] Yingkui Cao, Yanzhen Zou, Yuxiang Luo, Bing Xie, and Junfeng Zhao. 2018. Toward accurate link between code and software documentation. *Science China Information Sciences* 61, 5 (2018), 050105.

[17] Nuno Ramos Carvalho, José João Almeida, Pedro Rangel Henriques, and Maria João Varanda. 2015. From source code identifiers to natural language terms. *Journal of Systems and Software* 100 (2015), 117–128.

[18] Anna Corazza, Sergio Di Martino, and Valerio Maggio. 2013. LINSEN: An efficient approach to split identifiers and expand abbreviations. In *Proceedings of the International Conference on Software Maintenance (ICSM'13)*. 233–242.

[19] Andrea De Lucia, Massimiliano Di Penta, and Rocco Oliveto. 2011. Improving source code lexicon via traceability and information retrieval. *IEEE Transactions on Software Engineering* 37, 2 (2011), 205–227.

[20] Florian Deissenboeck and Markus Pizka. 2015. Concise and consistent naming: Ten years later. In *Proceedings of the 23rd International Conference on Program Comprehension (ICPC'15)*. 3.
[21] Eric Enslen, Emily Hill, Lori Pollock, and K. Vijay-Shanker. 2009. Mining source code to automatically split identifiers for software analysis. In *Proceedings of the IEEE International Working Conference on Mining Software Repositories (MSR'09)*. 71–80.
[22] J.-R. Falleri, Marianne Huchard, Mathieu Lafourcade, Clémentine Nebut, Violaine Prince, and Michel Dao. 2010. Automatic extraction of a WordNet-like identifier network from software. In *Proceedings of the International Conference on Program Comprehension (ICPC'10)*. 4–13.
[23] Asger Feldthaus, Todd Millstein, Anders Møller, Max Schäfer, and Frank Tip. 2011. Tool-supported refactoring for JavaScript. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA'11)*. 119–138.
[24] Xi Ge, Quinton L. DuBose, and Emerson Murphy-Hill. 2012. Reconciling manual and automatic refactoring. In *Proceedings of the 34th International Conference on Software Engineering (ICSE'12)*. 211–221.
[25] Latifa Guerrouj, Massimiliano Di Penta, Giuliano Antoniol, and Yann-Gael Gueheneuc. 2013. TIDIER: An identifier splitting approach using speech recognition techniques. *Journal of Software: Evolution and Process* 25, 6 (2013), 575–599.
[26] Latifa Guerrouj, Yann Gaël Guéhéneuc, Giuliano Antoniol, and Massimiliano Di Penta. 2012. TRIS: A fast and accurate identifiers splitting and expansion algorithm. In *Proceedings of the 19th Working Conference on Reverse Engineering (WCRE'12)*. 103–112.
[27] Emily Hill, David Binkley, Dawn Lawrie, Lori Pollock, and K. Vijay-Shanker. 2014. An empirical study of identifier splitting techniques. *Empirical Software Engineering* 19, 6 (Dec. 2014), 1754–1780.
[28] Johannes C. Hofmeister, Janet Siegmund, and Daniel V. Holt. 2018. Shorter identifier names take longer to comprehend. *Empirical Software Engineering* 24, 6 (2018), 1–27.
[29] Yanjie Jiang, Hui Liu, Jiahao Jin, and Lu Zhang. 2022. Automated expansion of abbreviations based on semantic relation and transfer expansion. *IEEE Transactions on Software Engineering* 48, 2 (2022), 519–537.
[30] Yanjie Jiang, Hui Liu, and Lu Zhang. 2019. Semantic relation based expansion of abbreviations. In *Proceedings of the Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'19)*. 131–141.
[31] Suntae Kim and Dongsun Kim. 2016. Automatic identifier inconsistency detection using code dictionary. *Empirical Software Engineering* 21, 2 (April 2016), 565–604.
[32] Dawn Lawrie and David Binkley. 2011. Expanding identifiers to normalize source code vocabulary. In *Proceedings of the International Conference on Software Maintenance (ICSM'11)*. 113–122.
[33] Dawn Lawrie, Henry Feild, and David Binkley. 2006. Syntactic identifier conciseness and consistency. In *Proceedings of the IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'06)*. 139–148.
[34] Guangjie Li, Hui Liu, and Ally S. Nyamawe. 2020. A survey on renamings of software entities. *ACM Computing Survey* 53, 2 (2020), Article 41, 38 pages.
[35] Jiechu Li, Qingfeng Du, Kun Shi, Yu He, and Jincheng Xu. 2018. Helpful or not? An investigation on the feasibility of identifier splitting via CNN-BiLSTM-CRF. In *Proceedings of the 30th International Conference on Software Engineering and Knowledge Engineering (SEKE'18)*. 175–214.
[36] Jiahui Liang, Weiqin Zou, Jingxuan Zhang, Zhiqiu Huang, and Chenxing Sun. 21. A deep method renaming prediction and refinement approach for Java projects. In *Proceedings of the International Conference on Software Quality, Reliability, and Security (QRS'21)*. 404–413.
[37] Bin Lin, Csaba Nagy, Gabriele Bavota, Andrian Marcus, and Michele Lanza. 2019. On the quality of identifiers in test code. In *Proceedings of the 19th International Working Conference on Source Code Analysis and Manipulation (SCAM'19)*. 204–215.
[38] Bin Lin, Simone Scalabrino, Andrea Mocci, Rocco Oliveto, and Michele Lanza. 2017. Investigating the use of code analysis and NLP to promote a consistent usage of identifiers. In *Proceedings of the International Working Conference on Source Code Analysis and Manipulation (SCAM'17)*. 81–90.
[39] Hui Liu, Qiurong Liu, Yang Liu, and Zhouding Wang. 2015. Identifying renaming opportunities by expanding conducted rename refactorings. *IEEE Transactions on Software Engineering* 41, 9 (2015), 887–900.
[40] Hui Liu, Qiurong Liu, Cristian Alexandru Staicu, Michael Pradel, and Luo Yue. 2016. Nomen est omen: Exploring and exploiting similarities between argument and parameter names. In *Proceedings of the International Conference on Software Engineering (ICSE'16)*. 1063–1073.
[41] Kui Liu, Dongsun Kim, Tegawendé F. Bissyandé, Taeyoung Kim, Kisub Kim, Anil Koyuncu, Suntae Kim, and Yves Le Traon. 2019. Learning to spot and refactor inconsistent method names. In *Proceedings of the 41st International Conference on Software Engineering (ICSE'19)*. 1–12.

[42] Siyuan Liu, Jingxuan Zhang, Jiahui Liang, Junpeng Luo, Yong Xu, and Chenxing Sun. 2021. CHIS: A novel hybrid granularity identifier splitting approach. In *Proceedings of the 28th Asia-Pacific Software Engineering Conference (APSEC'21)*. 192–201.

[43] Philip Mayer and Andreas Schroeder. 2014. Automated multi-language artifact binding and rename refactoring between Java and DSLs used by Java frameworks. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'14)*. 437–462.

[44] Christian D. Newman, Michael J. Decker, Reem Alsuhaibani, Anthony Peruma, Mohamed Mkaouer, Satyajit Mohapatra, Tejal Vishoi, Marcos Zampieri, Timothy Sheldon, and Emily Hill. 2022. An ensemble approach for annotating source code identifiers with part-of-speech tags. *IEEE Transactions on Software Engineering* 48, 9 (2022), 3506–3522.

[45] Son Nguyen, Hung Phan, Trinh Le, and Tien N. Nguyen. 2020. Suggesting natural method names to check name consistencies. In *Proceedings of the 42nd International Conference on Software Engineering (ICSE'20)*. 1372–1384.

[46] Jeffrey L. Overbey, Ralph E. Johnson, and Munawar Hafiz. 2016. Differential precondition checking: A language-independent, reusable analysis for refactoring engines. *Automated Software Engineering* 23, 1 (2016), 77–104.

[47] Jevgenija Pantiuchina, Fiorella Zampetti, Simone Scalabrino, Valentina Piantadosi, Rocco Oliveto, Gabriele Bavota, and Massimiliano Di Penta. 2020. Why developers refactor source code: A mining-based study. *ACM Transactions Software Engineering and Methodology* 29, 4 (2020), Article 29, 30 pages.

[48] Anthony Peruma, Mohamed Wiem Mkaouer, Michael J. Decker, and Christian D. Newman. 2018. An empirical investigation of how and why developers rename identifiers. In *Proceedings of the 2nd International Workshop on Refactoring (IWoR'18)*. ACM, New York, NY, 26–33.

[49] Anthony Peruma, Mohamed Wiem Mkaouer, Michael J. Decker, and Christian D. Newman. 2020. Contextualizing rename decisions using refactorings, commit messages, and data types. *Journal of Systems and Software* 169 (2020), 110704.

[50] Veselin Raychev, Martin Vechev, and Andreas Krause. 2015. Predicting program properties from "Big Code." In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium*. 111–124.

[51] Chanchal K. Roy and James R. Cordy. 2008. NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In *Proceedings of the 16th International Conference on Program Comprehension (ICPC'08)*. 172–181.

[52] Walter Savitch. 2004. *Java: An Introduction to Problem Solving and Programming* (4th ed.). Prentice Hall.

[53] Simone Scalabrino, Gabriele Bavota, Christopher Vendome, Mario Linares-Vasquez, Denys Poshyvanyk, and Rocco Oliveto. 2017. Automatically assessing code understandability: How far are we? In *Proceedings of the 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE'17)*. 417–427.

[54] Giuseppe Scanniello, Michele Risi, Porfirio Tramontana, and Simone Romano. 2017. Fixing faults in C and Java source code: Abbreviated vs. full-word identifier names. *ACM Transactions on Software Engineering and Methodology* 26, 2 (2017), Article 6, 43 pages.

[55] Hagen Schink, Martin Kuhlemann, Gunter Saake, and Ralf Lämmel. 2011. Hurdles in multi-language refactoring of hibernate applications. In *Proceedings of the 6th International Conference on Software and Database Technologies*. 129–134.

[56] Danilo Silva, Nikolaos Tsantalis, and Marco Tulio Valente. 2016. Why we refactor? Confessions of GitHub contributors. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'16)*. 858–870.

[57] Giriprasad Sridhara, Lori Pollock, and K. Vijay-Shanker. 2011. Automatically detecting and describing high level actions within methods. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE'11)*. 101–110.

[58] Takayuki Suzuki, Kazunori Sakamoto, Fuyuki Ishikawa, and Shinichi Honiden. 2014. An approach for evaluating and suggesting method names using n-gram models. In *Proceedings of the 22nd International Conference on Program Comprehension (ICPC'14)*. 271–274.

[59] Zhaopeng Tu, Zhendong Su, and Premkumar Devanbu. 2014. On the localness of software. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'14)*. 269–280.

[60] Xin Xia, Lingfeng Bao, David Lo, Zhenchang Xing, Ahmed E. Hassan, and Shanping Li. 2018. Measuring program comprehension: A large-scale field study with professionals. *IEEE Transactions on Software Engineering* 44, 10 (Oct. 2018), 951–976.

[61] Meng Yan, Xin Xia, Emad Shihab, David Lo, Jianwei Yin, and Xiaohu Yang. 2019. Automating change-level self-admitted technical debt determination. *IEEE Transactions on Software Engineering* 45, 12 (2019), 1211–1229.

[62] Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig. 2018. Learning to mine aligned code and natural language pairs from Stack Overflow. In *Proceedings of the International Conference on Mining Software Repositories (MSR'18)*. 476–486.

[63] Norihiro Yoshida, Takeshi Hattori, and Katsuro Inoue. 2010. Finding similar defects using synonymous identifier retrieval. In *Proceedings of the 4th International Workshop on Software Clones (IWSC'10)*. 49–56.
[64] Jian Zhou, Hongyu Zhang, and David Lo. 2012. Where should the bugs be fixed? More accurate information retrieval-based bug localization based on bug reports. In *Proceedings of the 34th International Conference on Software Engineering (ICSE'12)*. 14–24.

Received 1 April 2022; revised 12 February 2023; accepted 27 April 2023