On the Significance of Category Prediction for Code-Comment Synchronization

ZHEN YANG[‡] and JACKY WAI KEUNG, Department of Computer Science, City University of Hong Kong, China
XIAO YU, School of Computer Science and Artificial Intelligence, Wuhan University of Technology, China
YAN XIAO, School of Computing, National University of Singapore, Singapore
ZHI JIN, Key Laboratory of High Confidence Software Technologies, Ministry of Education, and School of Computer Science, Peking University, China
JINGYU ZHANG, Department of Computer Science, City University of Hong Kong, China

Software comments sometimes are not promptly updated in sync when the associated code is changed. The inconsistency between code and comments may mislead the developers and result in future bugs. Thus, studies concerning code-comment synchronization have become highly important, which aims to automatically synchronize comments with code changes. Existing code-comment synchronization approaches mainly contain two types, i.e., (1) deep learning-based (e.g., CUP), and (2) heuristic-based (e.g., HebCUP). The former constructs a neural machine translation-structured semantic model, which has a more generalized capability on synchronizing comments with software evolution and growth. However, the latter designs a series of rules for performing token-level replacements on old comments, which can generate the completely correct comments for the samples fully covered by their fine-designed heuristic rules. In this article, we propose a composite approach named **CBS** (i.e., **Classifying Before Synchronizing**) to further improve the code-comment synchronization performance, which combines the advantages of CUP and HebCUP with the assistance of inferred categories of **Code-Comment Inconsistent (CCI)** samples. Specifically, we firstly define two categories (i.e., heuristic-prone and non-heuristic-prone) for CCI samples and propose five features to assist category prediction. The samples whose comments can be correctly synchronized by HebCUP are heuristic-prone, while others are non-heuristic-prone. Then, CBS employs our proposed **Multi-Subsets**

[‡]This work was finally done during his visit to the Key Laboratory of High Confidence Software Technologies, Ministry of Education (Peking University).

This work is supported in part by the General Research Fund (GRF) of the Research Grants Council of Hong Kong, and the industry research funds of City University of Hong Kong (7005217, 9220097, 9220103, 9229029, 9229098, 9678149). The National Natural Science Foundation of China under Grant Nos. 62192731 and 61751210. The Singapore National Research Foundation and National University of Singapore through its National Satellite of Excellence in Trustworthy Software Systems (NSOE-TSS) under the Trustworthy Software Systems Core Technologies Grant (TSSCTG) NSOE-TSS2019-05. The Natural Science Foundation of Chongqing City (cstc2021jcyj-msxmX1115).

Authors' addresses: Z. Yang, J. W. Keung, and J. Zhang, Department of Computer Science, City University of Hong Kong, Tat Chee Avenue, Hong Kong, China, 999077; emails: zhyang8-c@my.cityu.edu.hk, Jacky.Keung@cityu.edu.hk, jzhang2297-c@my.cityu.edu.hk; X. Yu (corresponding author), School of Computer Science and Artificial Intelligence, Wuhan University of Technology, No. 122 Luoshi Road, Wuhan, Hubei, China, 430070; email: xiaoyu@whut.edu.cn; Y. Xiao, School of Computing, National University of Singapore, 21 Lower Kent Ridge Road, Singapore, 119077; email: dcsxan@nus.edu.sg; Z. Jin (corresponding author), Key Laboratory of High Confidence Software Technologies, Ministry of Education, and School of Computer Science, Peking University, No. 5 Yiheyuan Road, Beijing, China, 100871; email: zhijin@pku.edu.cn.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2023 Association for Computing Machinery.
1049-331X/2023/03-ART30 $15.00
https://doi.org/10.1145/3534117

Corrected Version of Record. V.1.1. Published May 11, 2023.

Ensemble Learning (MSEL) classification algorithm to alleviate the class imbalance problem and construct the category prediction model. Next, CBS uses the trained MSEL to predict the category of the new sample. If the predicted category is heuristic-prone, CBS employs HebCUP to conduct the code-comment synchronization for the sample, otherwise, CBS allocates CUP to handle it. Our extensive experiments demonstrate that CBS statistically significantly outperforms CUP and HebCUP, and obtains an average improvement of 23.47%, 22.84%, 3.04%, 3.04%, 1.64%, and 19.39% in terms of Accuracy, Recall@5, **Average Edit Distance (AED)**, **Relative Edit Distance (RED)**, BLEU-4, and **Effective Synchronized Sample (ESS)** ratio, respectively, which highlights that category prediction for CCI samples can boost the code-comment synchronization performance.

CCS Concepts: • **Software and its engineering → Software maintenance tools;**

Additional Key Words and Phrases: Code-comment synchronization, category classification, deep learning, heuristic rules

ACM Reference format:
Zhen Yang, Jacky Wai Keung, Xiao Yu, Yan Xiao, Zhi Jin, and Jingyu Zhang. 2023. On the Significance of Category Prediction for Code-Comment Synchronization. *ACM Trans. Softw. Eng. Methodol.* 32, 2, Article 30 (March 2023), 41 pages.
https://doi.org/10.1145/3534117

1 INTRODUCTION

Code comments give a clear natural language description for source code, which are critical for program comprehension and software maintenance [40, 45, 73]. As shown in Xia et al.'s study [91], software developers spend on average 58% of their time on program comprehension, where reading and comprehending code comments play a vital role. However, developers often neglect or forget to update the corresponding comments when code changes, which results in inconsistent or obsolete comments (i.e., bad comments) for their corresponding code [77, 78, 87]. Bad comments consume the effort of software developers in double-checking the code implementation and lead to the unintended injections of software bugs [35, 62, 80], which undoubtedly delay the project development and hinder software maintenance. If the comments can be correctly synchronized when software developers change the source code, the bad comments can be reduced or even avoided. Nevertheless, documenting source code is a labor-intensive work [20, 38], not to mention manually synchronizing comments with code changes. Therefore, the necessity of automating this process increases, which makes the code-comment synchronization a worthy research field.

Existing code-comment synchronization approaches can be divided into two categories: deep learning-based approach (e.g., CUP [53]) and heuristic-based approach (e.g., HebCUP [50]). The former constructs a **Neural Machine Translation (NMT)**-structured semantic model, while the latter designs a series of heuristic rules for token-level replacements on old comments. They both act on the code changes with bad comments left which are also referred to as **Code-Comment Inconsistent (CCI)** samples in this paper. However, CCI samples are complicated in practice, and the single type of approach is not enough to accurately synchronize all comments when their associated code is constantly evolving with various kinds of manners. For example, some tokens that need to be updated in comments do not appear in the corresponding code change contents, which makes it impossible for the heuristic-based approach to design hand-crafted heuristic rules to synchronize comments correctly [50]. Certain code snippets contain some high-frequency tokens that are not related to the code changes, which causes the deep learning-based approach to be distracted and ignore the key information of code changes during code-comment synchronization [24, 85, 96]. On the other hand, approaches of different kinds also carry heterogeneous

characteristics. Typically, deep learning-based approaches have a more generalized capability of synchronizing comments with software evolution and growth. And heuristic-based approaches can generate completely correct comments for CCI samples fully covered by their fine-designed heuristic rules.

Therefore, towards the state-of-the-art deep learning-based and heuristic-based approaches (i.e., CUP and HebCUP), we first experimentally verify their different capabilities on an open-source code-comment synchronization dataset, which is extracted from 1,496 Java repositories [53]. The dataset has been split into three parts, i.e., 80,591 samples in the training set, 8,827 samples in the validation set, and 9,204 samples in the testing set. Based on the empirical study in Section 3, we find that CUP and HebCUP indeed have their own unique advantages, and neither can dominate on all evaluation metrics, because these metrics assess code-comment synchronization approaches from different aspects. HebCUP can correctly synchronize more CCI samples than CUP does, therefore it defeats CUP in terms of Accuracy. In contrast, CUP has a more generalized capability due to its deep learning-based structure, causing it to defeat HebCUP in terms of **Average Edit Distance (AED)**, **Relative Edit Distance (RED)**, and BLEU-4. In addition, owing to its beam search technique, CUP also performs better in terms of Recall@5. We also find that both HebCUP and CUP have some distinctive CCI samples that the other side cannot handle but can be better solved by themselves. In particular, HebCUP prefers to handle CCI samples that their code changes contain more token replacements and fewer token insertions or deletions. However, CUP is not overly dependent on the edit actions of CCI samples. It is noteworthy that CUP outperforms HebCUP in terms of AED and RED, but has fewer CCI samples whose edit distances are reduced after the synchronization. This finding motivates us to propose a new evaluation metric, **Effective Synchronized Sample (ESS)** ratio, to complement AED and RED, which measures the ratio of CCI samples whose edit distances are reduced after being handled by synchronizers.

Upon the analysis above, we argue that combining the advantages of HebCUP and CUP can be a rational attempt to further boost the code-comment synchronization performance, which is still blank in this field and worthy to try. An intuitive solution is to design a classifier to predict the proneness of each CCI sample on synchronizers (i.e., either CUP or HebCUP) and assign each of them to one of the synchronizers according to the predicted proneness. Therefore, each approach is only responsible for the samples that they can handle better, and the overall performance of the code-comment synchronization can be further improved.

Nevertheless, there are some research challenges that need to be resolved: (1) How to accurately predict the proneness of a CCI sample in advance between the approaches of HebCUP and CUP? There is no previous study on the category prediction for CCI samples. Besides, it is also hard to define or represent the internal relation patterns between CCI samples and their model proneness. (2) How to deal with the class imbalance problem in the proneness prediction? Based on our preliminary statistics, comments of only 23.83% of CCI samples in the training set can be correctly synchronized by the HebCUP, which means the proneness prediction is a class imbalance problem. Simply adopting normal classifiers may bias the overall prediction performance, which is also a challenge that needs to be resolved.

This paper takes the first step towards constructing a composite approach named **CBS** (i.e., **Classifying Before Synchronizing)**, which combines the advantages of CUP and HebCUP with the assistance of inferred categories of CCI samples. We firstly define two categories for CCI samples, i.e., the samples that can be correctly synchronized by HebCUP are heuristic-prone, while others are non-heuristic-prone. In addition, we propose five features to facilitate the representation of internal relation patterns between CCI samples and their model proneness, thereby assisting category prediction. Then, since there are more non-heuristic-prone samples than heuristic-prone ones in the dataset, we propose a **Multi-Subsets Ensemble Learning (MSEL)** classification

algorithm to alleviate the class imbalance problem and construct the category prediction model. Next, CBS employs the trained classification model to predict the category of the new CCI sample. Finally, if the predicted category is heuristic-prone, CBS adopts HebCUP to synchronize comments of the sample, otherwise CBS assigns CUP to handle the sample.

We evaluate CBS and the two baselines (i.e., CUP and HebCUP) on the testing set in terms of Accuracy, Recall@5, AED, RED, BLEU-4, and ESS ratio. The experimental results demonstrate that CBS, on average, achieves an Accuracy score of 27.90%, a Recall@5 score of 35.67%, an AED score of 3.562, a RED score of 0.937, a BLEU-4 score of 73.16, and a ESS ratio of 35.85%. In addition, CBS outperforms HebCUP by 8.53% in terms of Accuracy, by 35.16% in terms of Recall@5, by 4.85% in terms of AED and RED, by 1.67% in terms of BLEU-4, and by 12.09% in terms of ESS ratio. CBS also performs better than CUP by 38.41%, 10.52%, 1.24%, 1.24%, 1.62%, and 26.69% in terms of the six evaluation metrics, respectively. Subsequently, we further discuss the performance of CBS with different base classifiers, the effect of the MSEL algorithm, and investigate the impact of the proposed five features. In summary, according to our extensive experiments and discussion, the category prediction for CCI samples can indeed boost the performance of code-comment synchronization via combining it with current approaches.

To the best of our knowledge, this is the first work that takes advantage of category prediction on the **Code-Comment Inconsistent (CCI)** samples to improve the performance of code-comment synchronization approaches. The main contribution of our work can be summarized as follows:

- We systematically investigate the current state-of-the-art code-comment synchronization approaches of heuristic-based and deep learning-based, which verifies the heterogeneous advantages of each approach and the existence of model proneness among CCI samples.
- We define two categories (i.e., heuristic-prone and non-heuristic-prone) for CCI samples and propose a CBS approach to further improve the performance of code-comment synchronization models, which demonstrates how category prediction of CCI samples can benefit the code-comment synchronization.
- We propose five original features to facilitate the representation of internal relation patterns between CCI samples and their model proneness, and design a Multi-Subsets Ensemble Learning (MSEL) algorithm to alleviate the class imbalance problem in category prediction.
- Extensive experiments are conducted to evaluate CBS, and a new evaluation metric named Effective Synchronized Sample (ESS) ratio is proposed to make the assessment of this work more comprehensive.
- Our research highlights that category prediction for CCI samples can make code-comment synchronization more practical in the real development scenario and sheds light on future directions for both industry and academia in this field.
- We open source the code of our CBS approach [92] to facilitate future research and application.

The remainder of this paper is organized as follows: Section 2 clarifies the background of code-comment synchronization, including the problem formalisation, the usage scenario, the state-of-art approaches, and a code-comment synchronization example. Section 3 empirically explores the existing code-comment synchronization approaches. Section 4 describes the methodology of our proposed CBS approach. Sections 5 and 6 present the experimental results for category prediction and code-comment synchronization, respectively. Section 7 explores the effect of the inside structures of our proposed approach. Section 8 discusses the failures of CBS and summarizes the implications of this work. Section 9 presents threats to validity. Section 10 briefly introduces related work. Finally, Section 11 concludes this paper.

2 BACKGROUND

In this section, we introduce the background of code-comment synchronization, including its problem formalization, usage scenario, state-of-the-art methods, and an example.

2.1 Problem Formalization

Code-comment synchronization aims to synchronize comments when the associated code changes. More formally, given a piece of code change, including a pre- and a post-change version of the code snippet, namely c and $c^{'}$, as well as a pre- and a post-change version of the comment associated with it, namely x and y, the objective of this research is to find a function f (i.e., a kind of code-comment synchronization approach), such that the post-change comment y can be correctly produced from f on condition of given c, $c^{'}$, and x as below.

$$y = f(c, c^{'}, x) \tag{1}$$

Hereon, the c, $c^{'}$, x, and y are referred to as old code, new code, old comment, and new comment (it is also named as reference comment in other studies), respectively. For each triple composed of c, $c^{'}$, and x, we refer to it as a **Code-Comment Inconsistent (CCI)** sample in this paper. The associated y is our target and is not available during the code-comment synchronization in practice. We cope with this problem by proposing a CBS approach to approximate f, and refer to each comment outputted from CBS as a synchronized comment $\hat{y}$.

2.2 Usage Scenario

We illustrate the usage scenario of CBS as follows:

Firstly, CBS can promptly and automatically help developers synchronize the associated comment when making a code change. If the synchronized comment is correct or partially correct, it can greatly reduce the workloads of developers on manually editing comments and make them concentrate more on the code implementation of transaction logic, thereby improving productivity. Besides, this process also effectively avoids the obsolete comments that are ignored to update by developers during their programming.

Secondly, CBS is also useful in repairing the existing bad comments. For example, when developers are programming based on others' code, they may encounter code snippets with bad comments, which will mislead their understanding of the code's intention. In this case, developers can adopt a bad comment detector, e.g., the tool proposed by Liu et al. [51], to identify comments that need to be synchronized, then CBS can be applied to them to automatically repair those comments and make them synchronous with their associated code snippets.

The usage of CBS can also be quite frequent. According to the study of Kolassa et al. [43] on 11,143 open-source projects with a total of 8,705,118 commits by 47,548 committers, over 50% of developers commit twelve times daily. Besides, we also conducted a preliminary analysis on the 1,496 Java projects in this paper, and we found that each commit is related to, on average, 2.2 method-level code changes. Hence, CBS, as a method-level code-comment synchronizer, can be utilized by developers very often.

2.3 Current Code-Comment Synchronization Approaches

CUP and HebCUP are two representative state-of-the-art code-comment synchronization approaches. Here, we introduce them as below:

- **CUP** is a deep learning-based model proposed by Liu et al. [53]. They describe the code-comment synchronization task as a machine translation process and design an LSTM-based

Sequence-To-Sequence (Seq2Seq) variant to deal with the mapping from old comments and code changes to new comments. Specifically, the semantics of code change sequences and old comment sequences are progressively extracted through the layers of embedding, co-attention, and modeling. Afterward, the high-level representation of code change sequences and old comment sequences are decoded with the previously generated tokens of new comments via the dot-production attention mechanism to generate a new comment token at each time step.

- **HebCUP** is a heuristic-based approach proposed by Lin et al. [50]. They put forward a series of heuristic rules by empirically studying CUP successful and failure cases. HebCUP can successfully handle most code-indicative changes, whose changed sub-tokens or tokens in comments can be found from the corresponding code change contents [50]. Specifically, HebCUP first aligns the sub-tokens in code change sequences according to its sub-token alignment algorithm and constructs a series of replacement pairs based on the alignment results. Each replacement pair can be represented as a format of *key-value*, where the *key* is the old token while the *value* is the potential new token for replacement. Finally, HebCUP adopts *keys* to match tokens in old comments and conduct the updating with *values*, thereby conducting the synchronization for comments.

2.4 An Example of Code-Comment Synchronization

We present a code-comment co-change instance in Listing 1, where the comment in red is the old comment describing the functionality of the old code, the comment in green is the new comment describing the functionality of the new code, the code in red is the deleted statement, the code in green is the newly added statement, and the other code is no-change statements. Old code is composed of no-change statements and deleted statements while new code is composed of no-change statements and newly added statements. Given a triple of old code, new code, and old comment, a code-comment synchronizer (e.g., CUP and HebCUP) is able to provide a comment suggestion of describing the new code, and we refer to such suggestion as a synchronized comment. Taking the Listing 1 as an example, CUP will generate a comment suggestion of "Return the PIM Interface" for it. Subsequently, this synchronized comment will be evaluated in terms of the metrics we illustrate in Section 3.2 via making a series of comparisons between itself and the reference comment (i.e., "Return the ONOS Interface"), thereby assessing the quality of each synchronized comment.

```
1 - /*Get the PIM Interface.*/
2 + /*Return the ONOS Interface.*/
3     public Interface getInterface () {
4 -             return theInterface;
5 +             return this.onosInterface;
6           }
```

Listing 1. A code-comment co-change instance.

3 EXPLORING EXISTING CODE-COMMENT SYNCHRONIZATION METHODS

In this section, we experimentally investigate the performance of CUP and HebCUP to explore their respective advantages and their heterogeneous characteristics.

Old Code Sequence	'public', 'Interface', 'get', '<con>', 'Interface', '()', '{', 'return'	'the'	'<con>', 'Interface', '<con>', ';', '}'
New Code Sequence	'public', 'Interface', 'get', '<con>', 'Interface', '()', '{', 'return'	'this', '<con>', '.', '<con>', 'onos'	'<con>', 'Interface', '<con>', ';', '}'
Action Sequence	equal	replace	equal

Fig. 1. The code change sequence of the code-comment co-change instance on sub-sequence-level.

3.1 Dataset

The dataset we adopt in this work is extracted from 1,496 Java repositories hosted on GitHub [6], where most of them are famous projects, such as the Nomulus [7] of Google, the Hive [3] of Apache, and Fresco [5] of Facebook. The dataset was first built by Wen et al. [87], and first applied in the code-comment synchronization field by Liu et al. [53]. Subsequently, Lin et al. [50] further cleaned the dataset. According to Liu et al. [53], they deduplicated the dataset and arranged it in ascending order according to the commit creation time within each project. The first 80% of commits were put into the training set, and the remaining 20% of commits were shuffled and evenly split into the validation and test set, thereby avoiding the data leakage problem. Subsequently, Lin et al. [50] further cleaned the dataset and obtained 80,591, 8,827, and 9,204 code-comment co-change instances in the training, validation, and testing sets, respectively. For each instance, the old code, new code, and old comment constitute a CCI sample in practice, and the new comment is regarded as the reference comment for synchronization. Code and comments are tokenized to sequences, and their tokens are broken into sub-tokens based on camel casing and snake casing. Besides, "*<con>*" is also inserted to join the sub-tokens for future recovery. For example, the token "*getInterface*" is tokenized to two sub-tokens "*get*" and "*Interface*", and "*<con>*" is inserted to join the two sub-tokens.

In particular, according to Liu et al. [53] who construct the original version of this dataset, if the word-level Levenshtein distance [57] of a comment pair (i.e., an old comment and its associated new comment) is larger than the old comment's length and 5, they regarded this pair as a rewrite instead of an update, where the word-level Levenshtein distance is minimum word edits required to change a sentence into the other. This makes sense, because if the distance of a comment pair is too large, it has a great possibility that there is little relation between the pre- and post-change version of code and comments, which should better use code summarization approaches to generate comments from scratch. On the other hand, if we still use code-comment synchronization approaches to deal with them, the synchronized comments may be not as good as we expected because the relation between old and new code (comment) has almost disappeared. Therefore, setting some restrictions on the distances of comment pairs is reasonable. In addition, according to the study of Liu et al. [53], before they conducted this filtering procedure, they collected a dataset with 242,649 samples. After the filtering, there still remains 147,844 samples which still account for over 60% of the original samples. Therefore, these samples with relatively small changes indeed occur very often in code changes, whereas the larger changes are the minorities. Thus, this dataset is practical.

Table 1 presents the statistics of the dataset we adopted, where Length is the number of sub-tokens in each type of data sequence, and **Number of Changed Sub-tokens (NCS)** records the number of sub-tokens that are modified between the old and new code (comments). According to the statistical results in Table 1, we find that the Lengths of code (comments) among each part of the dataset are similar on both the mean and standard deviation. Besides, the same phenomenon can also be observed in the NCS. It indicates there is no significant difference in the data distributions among the training, validation, and testing sets.

Table 1. The Statistics of the Datasets

Dataset	Type	Length			NCS		
		Mean	Standard Deviation	Median	Mean	Standard Deviation	Median
Train	Old Code	91.22	84.42	60.0	34.73	57.49	11.0
	New Code	91.26	84.98	58.0			
	Old Comment	14.43	7.34	13.0	3.81	3.01	3.0
	New Comment	14.47	7.63	13.0			
Validation	Old Code	89.60	82.61	59.0	32.73	54.66	12.0
	New Code	88.77	82.66	57.0			
	Old Comment	15.16	7.57	14.0	3.83	3.03	3.0
	New Comment	15.21	7.83	14.0			
Test	Old Code	97.61	86.92	64.0	36.85	63.73	12.0
	New Code	97.44	87.94	63.0			
	Old Comment	15.37	7.52	14.0	3.97	3.11	3.0
	New Comment	15.29	7.77	14.0			

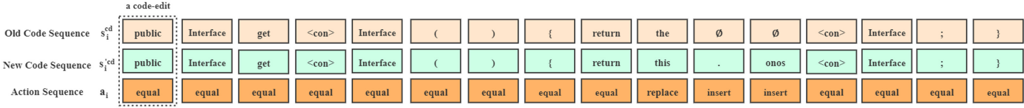

Fig. 2. The code change sequence of the code-comment co-change instance on element-level.

To better represent the code changes during the synchronization, Liu et al. [52, 53] adopted a Python *diff* tool namely SequenceMatcher [4] to align each old code sequence and its associated new code sequence. The SequenceMatcher is able to identify four edit actions, including "equal", "replace", "insert", and "delete" at the sub-sequence-level between two sequences. For example, Listing 1 presents a code-comment co-change instance. After breaking the old code, and new code of this sample into sub-tokens, SequenceMatcher can align old and new code sequences of Listing 1, and present results as shown in Figure 1. Subsequently, to convert the alignment from sub-sequence-level to element-level, Liu et al. conducted a series of operations: (1) for sub-sequence matches marked as "equal", they keep them intact, (2) for those marked as "insert" and "delete", they directly fill up empty tokens "∅" for the shorter ones to align them, and (3) for those marked as "replace", since the lengths of two sub-sequences in each matched pair may not be equal, they align them from the head or tail according to the similarity of their first and last tokens. In detail, if the similarity between the first tokens of the two sub-sequences is larger than that between the last tokens, they align the two sub-sequences from their head and add empty tokens "∅" to fill up the short one, otherwise, they align the two sub-sequences from the tail and add empty tokens "∅" accordingly. The similarity is computed by *quick_ratio* which is a built-in function of SequenceMatcher. For example, towards the sole "replace" edit action between the sub-sequence (i.e., ["the"]) of the old code and the sub-sequence (i.e., ["this", "<con>", ".", "<con>", "onos"]) of the new code shown in the Figure 1, sub-token "the" is closer to sub-token "this" than to sub-token "onos" according to the similarity score computed by *quick_ratio*. Thus, sub-sequence ["the"] aligns sub-sequence ["this", "<con>", ".", "<con>", "onos"] from head, and empty tokens "∅" fills up the shorter sub-sequence ["the"] in the old code. Figure 2 is the element-level aligned code change sequence processed by the above procedure based on Listing 1. Each element, i.e., code-edit, in the code change sequences is a triple $<s_i^{cd}, s_i'^{cd}, a_i>$, where s_i^{cd} is the ith sub-token in the old code, $s_i'^{cd}$ is the ith sub-token in the new code, and a_i denotes the edit action that converts s_i^{cd} to $s_i'^{cd}$, which can be "equal", "replace", "insert", and "delete" as we mentioned above. If a_i is "insert" ("delete"), s_i^{cd} ($s_i'^{cd}$) will be the empty token ∅.

3.2 Evaluation Metrics for Code-Comment Synchronization

Following Liu and Lin et al.'s works [50, 53], we adopt **Accuracy**, **Recall@5**, **AED**, **RED**, and **BLEU-4** to evaluate the performance of the two code-comment synchronization approaches.

- **Accuracy** measures the capability of code-comment synchronization approaches on *correct synchronizations*. The term *correct synchronizations* means the synchronized comments are identical with the reference comments, which also means CCI samples are correctly synchronized. Specifically, Accuracy is the percentage of CCI samples where *correct synchronizations* can be generated on the first attempt. More formally, it can be defined as below:

$$Accuracy = \frac{N_{correct}}{N}, \tag{2}$$

  where $N_{correct}$ is the number of *correct synchronizations*, and N is the number of testing samples. Thus, the higher the Accuracy, the better the performance of code-comment synchronizers. When judging the identicalness between synchronized comments and reference comments, Liu et al. [53] ignore the punctuations at the end of the comments, while Lin et al. [50] ignore the case and all punctuations in comments. Since both punctuation and case will not affect developers' understanding of comments too much, we uniformly adopt the code implementation of Lin et al. to calculate the Accuracy of synchronizers in our study.
- **Recall@5** is similar to Accuracy but allows an approach to conduct five attempts for each CCI sample. If any one of the synchronized comments is a *correct synchronization*, it considers the approach can successfully handle this sample [53]. Likewise, the higher, the better. This metric is also useful and practical because providing correctly synchronized comments in limited (e.g., five) candidates also saves the efforts of developers and offers alternative options.
- **AED** is the average edits that developers need to perform in order to change the synchronized comments to the reference comments after being handled by code-comment synchronizers, which indicates the distance between the synchronized comments to the reference comments. Therefore, the smaller the AED, the better the performance of code-comment synchronizers. More formally, AED can be defined as below:

$$AED = \frac{1}{N}\sum_{k=1}^{N} ED(\hat{y}^{(k)}, y^{(k)}), \tag{3}$$

  where ED is the edit distance (i.e., word-level Levenshtein distance [57]) between $\hat{y}^{(k)}$and y^k, $\hat{y}^{(k)}$ represents the kth synchronized comment, y^k denotes the kth reference comment, and N is the number of the testing samples. Since Lin et al. [50] did not publish the source code of calculating AED, we adopt the code implementation of Liu et al. [53] except that we ignore the case.
- **RED** is similar to AED, but presents the average of relative edit distance. In detail, it measures the ratio of AED between synchronized comments ($\hat{y}$s) and reference comments (ys) to the AED between old comments (xs) and reference comments (ys), which indicates to what extent an approach can release the burden of developers from a manual update: the smaller, the better.

$$RED = \frac{1/N\sum_{k=1}^{N} ED(\hat{y}^{(k)}, y^{(k)})}{1/N\sum_{k=1}^{N} ED(x^{(k)}, y^{(k)})}, \tag{4}$$

- **BLEU-4** is a universally applied precision-oriented evaluation metric in code summarization [17, 34, 93] and code-comment synchronization [53] fields, which measures the quality of

automatically generated sentences by computing the n-gram precision of a generated sentence to the reference sentence [61]: the higher, the better. More formally, it can be defined as follows:

$$BLEU - N = BP * exp\left(\sum_{n=1}^{N} w_n log\, p_n\right), \tag{5}$$

$$BP = \begin{cases} 1, & if \quad candi > ref \\ e^{1-\frac{ref}{candi}}, & if \quad candi \leq ref, \end{cases} \tag{6}$$

where p_n represents the ratio of sub-sequences of the length of n in candidated synchronized comments that also appear in the reference comments, BP represents the brevity penalty, $candi$ and ref represent the length of the candidated synchronized comments and reference comments, $w_n = \frac{1}{N}$, and $N = 4$ in this work is to reflect the weighted sum from 1 to 4.

To sum up, since Accuracy and Recall@5 only count the proportion of samples that are absolutely correctly synchronized, their evaluation results reflect the correct synchronization capability of approaches. However, AED, RED, and BLEU-4 compute the average similarity of the overall samples between synchronized comments and their corresponding reference comments. Thus, AED, RED, and BLEU-4 evaluate the general synchronization capability of approaches.

3.3 Experimental Design

In this section, first of all, we assess the performance of CUP and HebCUP on the same validation set and unified evaluation criteria. In detail, both CUP and HebCUP are constructed based on the source code published in their respective papers [50, 53]. CUP is trained from scratch on the training set then evaluated on the validation set, while HebCUP is directly evaluated on the validation set due to its training-free characteristic. With respect to the initial training parameters, hyperparameters, and training strategies of CUP, we fixed them according to the setting in its original paper [53], such that the re-trained CUP can reproduce its original performance. Afterward, the outputs of CUP and HebCUP on the validation set are uniformly evaluated via the metrics we illustrated in Section 3.2.

After obtaining and evaluating the outputs of CUP and HebCUP, we further deeply analyze the distributions of samples that are correctly synchronized within the first attempt (i.e., Accuracy) and five attempts (i.e., Recall@5) to explore the different advantages of CUP and HebCUP on *correct synchronization*. In addition, we also dig deeper for the distribution of samples in terms of edit distance (i.e., concerning the AED and RED) to explore the heterogeneous characteristics of HebCUP and CUP on general synchronization capability. Finally, for samples whose edit distances are reduced after the synchronization, we study them further to explore whether there are potential regularities in the sample preferences of approaches.

3.4 Results and Discussion

Table 2 presents the performance of CUP and HebCUP on the validation set under the unified evaluation metrics.

(1) As can be seen, HebCUP achieves a score of 25.60% in terms of Accuracy, 25.99% in terms of Recall@5, 3.68 in terms of AED, 1.009 in terms of RED, and 70.94 in terms of BLEU-4. CUP achieves a score of 21.75%, 30.50%, 3.45, 0.944, and 72.09 in terms of the five evaluation metrics, respectively. More specifically, CUP outperforms HebCUP by 17.35% in terms of Recall@5, 6.25% in terms of AED and RED, and 1.62% in terms of BLEU-4. On the contrary, HebCUP performs better than CUP in terms of Accuracy by 17.70%.

Table 2. The Code-Comment Synchronization Performance of HebCUP and CUP on the Validation Set

Approach	Accuracy (%)	Recall@5 (%)	AED	RED	BLEU-4
HebCUP	**25.60 [2260/8827]**	25.99	3.68	1.009	70.94
CUP	21.75 [1920/8827]	**30.50**	**3.45**	**0.944**	**72.09**

$^{\Phi}$The number of *correct synchronizations* are presented in square brackets, where the number of total validation samples is 8,827.

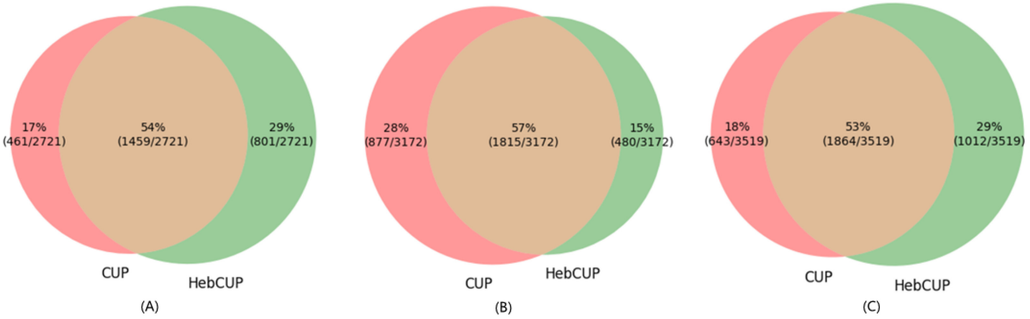

Fig. 3. (A) The Venn graph for the distributions of samples with correctly synchronized comments after being handled by CUP and HebCUP within the first attempt, (B) The Venn graph for the distributions of samples with correctly synchronized comments after being handled by CUP and HebCUP within five attempts, (C) The Venn graph for distributions of samples with $ED_diff < 0$ after being handled by CUP and HebCUP.

(2) It is noticeable that the Accuracy and Recall@5 on the validation set of CUP in our experiment are higher than those recorded in the paper of Lin et al. [50]. This is because Lin et al. did not uniformly apply the implementations of their evaluation metrics to CUP, and their implementations of the evaluation metrics for HebCUP are looser than those in the CUP paper (details refer to Section 3.2). Therefore, owing to such kind of unfair comparison, the performance of CUP in terms of Accuracy, Recall@5, AED, and RED are lower than those of HebCUP in the paper of Lin et al. [50]. However, as shown in Table 2, neither of the two approaches can dominate on all evaluation metrics.

To dig deeper into the performance of CUP and HebCUP, we plot three Venn graphs to describe the sample distributions related to the metrics of Accuracy, Recall@5, AED, and RED in Figure 3 and list our findings as the following:

(1) According to Figure 3(a), in totally 2,721 samples that can be correctly synchronized by CUP or HebCUP, CUP can correctly synchronize comments for 461 (17%) samples that HebCUP cannot handle. On the contrary, the latter can correctly synchronize comments for 801 (29%) samples that the former cannot handle. Besides, 1,459 (54%) samples can be correctly handled by both CUP and HebCUP.

(2) Nevertheless, if we expand the evaluation to five candidated synchronized comments, Figure 3(b) illustrates that CUP can correctly synchronize more distinctive samples (28%) than HebCUP (15%). This is due to the usage of beam search in the neural semantic model of CUP, which helps it infer a limited number of best predictions by approximately maximizing the conditional probability [23]. Whereas, the candidates of HebCUP are generated by permutation and combination within the replacement pairs, leading to a lack of diversity in most candidates. (3) Subsequently, we define an Equation (7) to describe edit distance difference (ED_diff) before

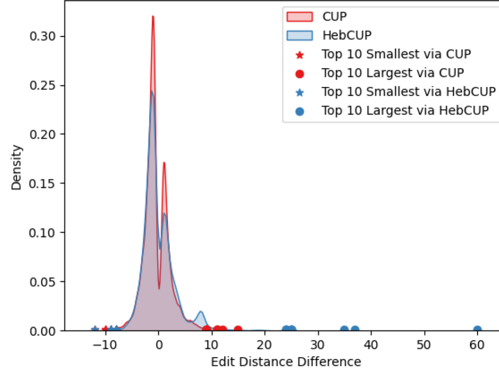

Fig. 4. The density plot for the sample distributions of $ED_diff \neq 0$ after being handled by CUP and HebCUP.

and after the code-comment synchronization.

$$ED_diff = ED(\hat{y}, y) - ED(x, y), \tag{7}$$

where $ED(\hat{y}, y)$ denotes the edit distance from the synchronized comment ($\hat{y}$) to the reference comment (y) while $ED(x, y)$ denotes the edit distance from the old comment (x) to the reference comment (y). For example, if $ED_diff < 0$, it represents the edit distance of a sample is reduced, that is to say, comparing with the old comment, the synchronized comment approaches closer to the reference comment, thus the code-comment synchronizer shows a positive effect; if $ED_diff > 0$, it illustrates the edit distance increases and the code-comment synchronizer has a negative effect on the sample; and if $ED_diff = 0$, it tells us the synchronizer has no effect. With the aid of ED_diff, Figure 3(c) demonstrates that the edit distances of 29% distinctive samples are reduced after being handled by HebCUP, while the edit distances of 18% distinctive samples are reduced by CUP.

It is noteworthy that CUP outperforms HebCUP in terms of AED and RED but contributes fewer edit distance reduced samples. In order to clearly explain this phenomenon, we plot samples of $ED_diff \neq 0$ in Figure 4 to more clearly present the distributions of samples whose edit distances are changed after being handled by CUP and HebCUP, where the red (blue) points represent the samples with the top 10 largest ED_diff values processed by CUP (HebCUP), and the red (blue) stars represent the samples with top 10 smallest ED_diff values processed by CUP (HebCUP).[1] We find that although HebCUP contributes more edit distance reduced samples, it produces more edit distance increased samples, and the edit distance of these samples increase even more.[2] Therefore, CUP outperforms HebCUP in terms of AED and RED. The above analysis inspires us that AED and RED indeed can describe the model performance concerning the edit distance over the entire samples on the sub-token-level, but they cannot measure the edit distance from the perspective of individual quantity (i.e., sample-level), such as counting the proportion of samples whose edit distances are reduced after being handled by synchronizers. In particular, approaches with better AED and RED may also have fewer samples whose edit distances are reduced after the synchronization as shown in the above comparison between CUP and HebCUP, which is also the issue

[1]The proportions of $ED_diff \neq 0$ for CUP and HebCUP are 43.94% and 52.71%, respectively, indicating the fact that both CUP and HebCUP indeed have a number of samples that cannot implement any synchronization for them. Nevertheless, this phenomenon does not affect the improvement of our proposal based on CUP and HebCUP.
[2]HebCUP: 1,776 samples with $ED_diff > 0$, where the edit distance increases 3.106 on average; CUP: 1,372 samples with $ED_diff > 0$, where the edit distance increases 2.103 on average.

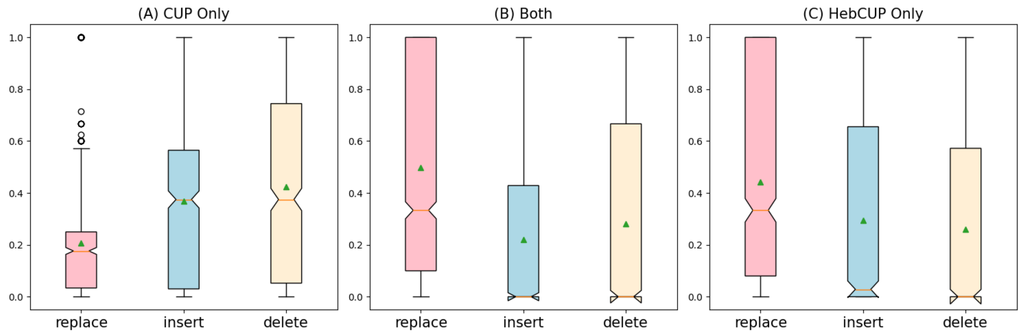

Fig. 5. The preferences of HebCUP and CUP on samples with different percentages of "replace", "insert", and "delete".

that the previous studies ignore. Therefore, in Section 6, we propose a new evaluation metric as complementation for AED and RED.

Afterwards, we further conduct a statistical analysis on the samples whose edit distances are reduced after the synchronization in Figure 5, where all these samples are partitioned into three kinds, i.e., (A) edit distances can be reduced by HebCUP only, (B) edit distances can be reduced by both approaches, and (C) edit distances can be reduced by CUP only. For each part of the samples, we show the percentages of their edit actions (i.e., "replace", "insert", and "delete") in code changes via box plots, where the green triangles represent the mean values and the orange lines represent the median values. As can be seen, HebCUP extremely prefers to handle samples with a high percentage of "replace" action, and around half of those samples do not have any edit actions of "insert" and "delete" according to Figure 5(b) and (c). The mechanism of HebCUP is exploiting replacement pairs to conduct token-level updates on old comments. On the other hand, according to Figures 5(a), we find that samples with edit actions of "insert" and "delete" account for a higher percentage compared with those with "replace" action. However, in Figure 5(b), samples with "replace" action account for the greatest proportion. Edit distances of samples in both figures can be reduced by CUP, but present quite different distributions over edit actions, showing that CUP is not overly dependent on the edit actions, like HebCUP, because it is driven by deep learning.

Conclusion

(1) HebCUP outperforms CUP in terms of Accuracy, while CUP produces better results in Recall@5, AED, RED, and BLEU-4.
(2) Both HebCUP and CUP have some distinctive samples that the other one cannot handle but can be better solved by themselves.
(3) HebCUP prefers to handle samples having more edit actions of "replace" and fewer actions of "insert" and "delete" in their code changes whereas CUP is not overly dependent on the edit actions like HebCUP during the code-comment synchronization.

4 METHODOLOGY

This section interprets the methodologies of CBS.

4.1 Motivation

According to the statistics and analysis in Section 3.4, we have verified the heterogeneous characteristics of CUP and HebCUP, including their superiority on different metrics and preferences

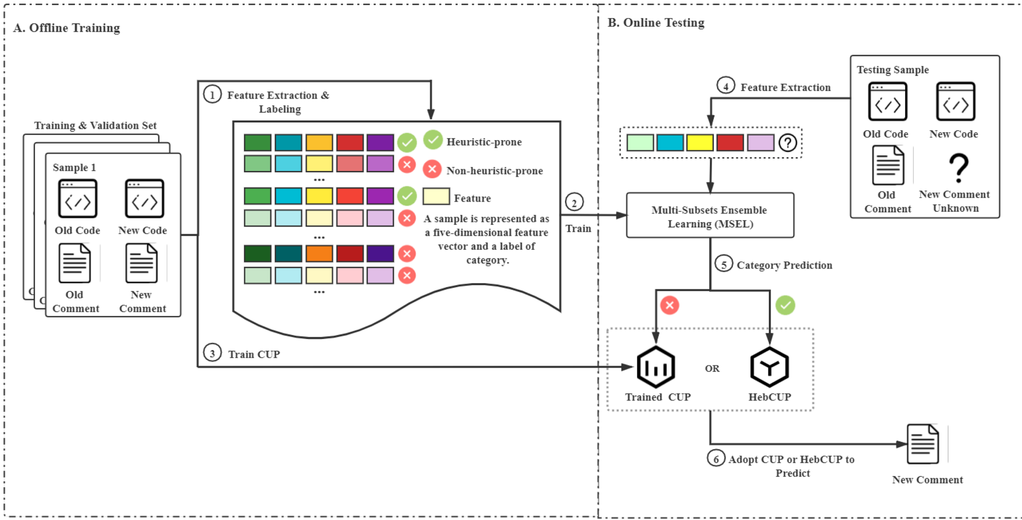

Fig. 6. The overview of the CBS approach.

on different kinds of samples, which are originated from the different properties and mechanisms of the two approaches. Therefore, it is a great possibility that combining these two kinds of approaches can be beneficial and reach a better result in the code-comment synchronization task. Due to this, we propose a Classifying Before Synchronizing (CBS) approach, which predicts the proneness of CCI samples on approaches (i.e., HebCUP and CUP) first, then assign samples to the most suitable approach to conduct the comment synchronization with code changes. Theoretically, the code-comment synchronization results of CBS approach heavily depend on the performance of category prediction for the proneness of CCI samples. If the prediction for the proneness is wrong, CCI samples will be assigned to the wrong approach, and the final synchronization result will be awful. Hence, our methodology mainly focuses on the following problems: (1) How to represent the internal relation patterns between CCI samples and their proneness to facilitate the category prediction. (2) The class imbalance problem during the category prediction is also an imperative issue that needs to be solved.

4.2 Approach Overview

Figure 6 shows the overall framework of our proposed CBS approach, which consists of two stages with multiple steps. For the offline training stage, we firstly extract the five predefined features from the training and validation sets, and label the samples as heuristic-prone or non-heuristic-prone in Step 1; In Step 2, exploiting the features we extract from the previous step, we train a Multi-Subsets Ensemble Learning (MSEL) classification model for category prediction of CCI samples. Besides, the MSEL classification model also alleviates the class imbalance problem during the category prediction due to its ensemble technique; Finally, we exploit the training and validation sets to train the CUP as the code-comment synchronizer for non-heuristic-prone samples in Step 3. Until now, the CBS has prepared for online testing as shown in the left-hand side of Figure 4.2. For the online testing stage, we firstly extract the same features from the testing sample in Step 4; Then, we employ the trained MSEL classification model (as mentioned in Step 2) to predict the category of the testing sample in Step 5; Finally, in Step 6, the predicted category is heuristic-prone, we adopt HebCUP to synchronize the comment of the sample, otherwise, we assign CUP to handle the sample.

In order to illustrate our approach more clearly, we present a running example here in Figure 7 to show how CBS handles CCI samples in practice via combining CUP and HebCUP, and list two CCI samples for illustration, the one is from Apache Wicket [1] which is a sample can be correctly synchronized by CUP, the another one is from the Google Nomulus [2], which can be correctly synchronized by HebCUP. For code changes, we highlight the statements that are deleted in red, and for those that are newly added, we highlight them in green. Besides, we also circle the specific changed regions in both code and comments for the convenience of illustration. As can be seen, regardless of any CCI samples, their five features will be extracted first, then the MSEL classification model will identify the categories of samples (i.e., Phase 1: Category Prediction), then allocate them to the corresponding code-comment synchronizers to generate new versions of comments and replace their old obsolete comments (i.e., Phase 2: Code-Comment Synchronization).

More specifically, Sample 1 is to fix the "createSessionFolder" value as "true" inside the method, thus the content of "and createSessionFolder is true" which is one of the conditions to create a new folder in comment should be removed. However, Sample 1 has two parts of changes in the code, the one is the deletion of ",", "boolean", and "createSessionFolder" in Line 5 and the other one is a replacement of "createSessionFolder" with "true" in Line 8-9. The two kinds of operations for "createSessionFolder" make HebCUP hardly tell what should be done for "createSessionFolder" in the old comment. More importantly, the tokens that need to be deleted in the old comment include contents that are not shown in the code changes, such as "and" and "is", thus it is impossible to design heuristic rules. On the contrary, CUP can successfully handle this sample for its learning and comprehension to the code and comment semantics. In this case, MSEL classification model correctly predicts the category (i.e., non-heuristic-prone) of Sample 1 and allocates it to CUP, thereby we obtain the *correct synchronization* for Sample 1, and list the result in the left-bottom corner of Figure 7. For Sample 2, it is going to change the path of the file in code, i.e., from "javatests/%s/testdata/%s" to "src/test/resources/%s/%s" in Line 8-9, and HebCUP is able to extract the replacement pairs between *key*: "javatests" and *value*: "src/test/resources", as well as *key*: "testdata" and *value*: "∅". As such, HebCUP exactly replaces the "javatests" with "src/test/resources" on the old comment, and achieves a *correct synchronization*. Nevertheless, CUP fails to conduct the *correct synchronization*, and its generated result is "Returns the "real" location of the file loaded by the other commands, starting from test/test.". Obviously, it repeatedly generates the word "test", which is a very high-frequency word, ranks 156 out of 44,577 in the whole vocabulary, and neglects the correct words, such as "src" and "resources". In this case, the MSEL classification model correctly predicts the category (i.e., heuristic-prone) of Sample 2 and allocates it to HebCUP, thereby we obtain the *correct synchronization* for it, and list the result in the right-bottom corner of Figure 7. Via the procedure we mentioned above, CBS can effectively exploit the advantages of both CUP and HebCUP, thereby continue boosting the performance of code-comment synchronization. We illustrate each part of CBS in detail in the following sections.

4.3 Category Definition and Partition Criteria

In this paper, we categorize CCI samples as heuristic or non-heuristic based on the synchronization results of HebCUP. In other words, for those samples that can be correctly synchronized by the HebCUP, we label them as heuristic-prone; otherwise, we label them as non-heuristic-prone. We adopt the synchronization results of HebCUP as the labeling criterion for a variety of reasons: (1) The labeling results of the deep learning-based approach, i.e., CUP, will change more or less for its randomness in training. The category prediction is hard to conduct with nonstationary labels. (2) Code-comment synchronization guided by hand-crafted rules is explainable, thus, it is relatively easier to represent the internal relation patterns between CCI samples and their proneness. According to the above definition and partition criterion, we label the categories for CCI

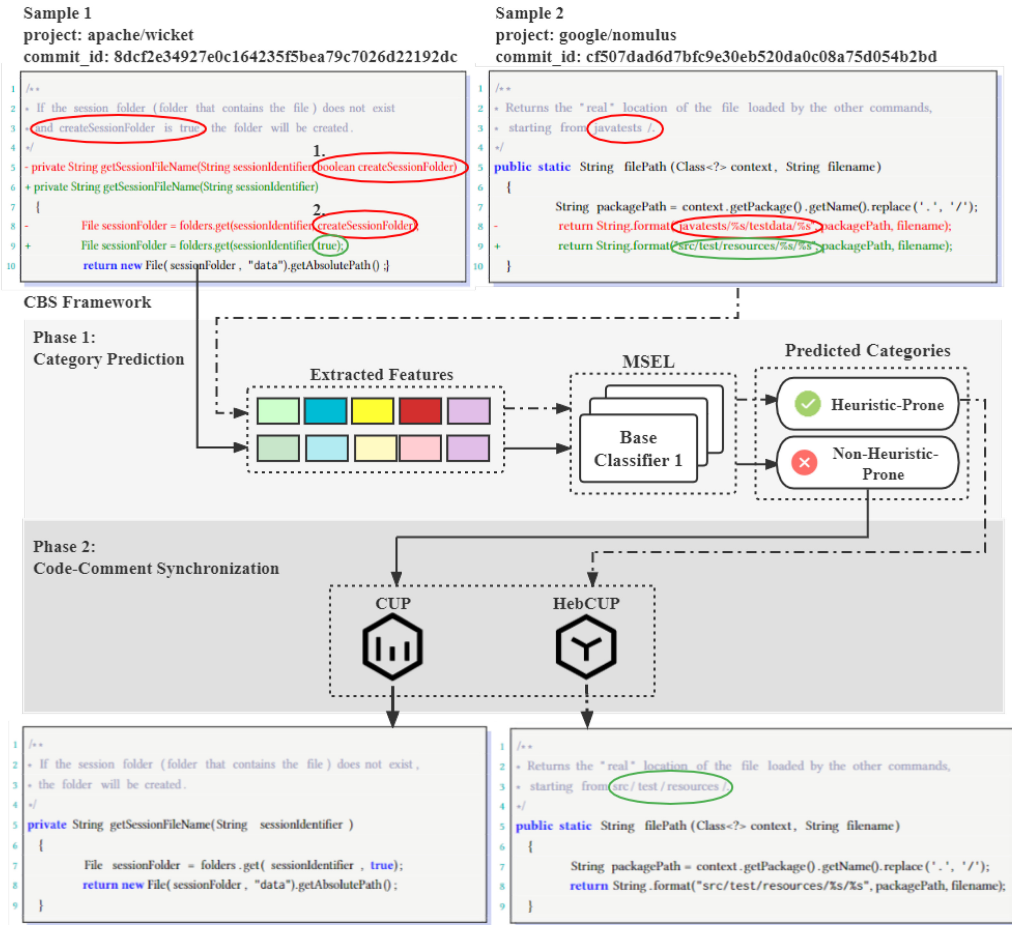

Fig. 7. A running example to show how CBS works.

Table 3. Category Distribution of Code-comment Inconsistent Samples in the Training and Validation Sets

Dataset	Category	Count	Proportion
Training Set	Heuristic-Prone	19,202	23.83%
	Non-Heuristic-Prone	61,389	76.17%
Validation Set	Heuristic-Prone	2,260	25.60%
	Non-Heuristic-Prone	6,567	74.39%

samples on the whole training and validation set. Table 3 demonstrates the category distribution of CCI samples on training set and validation set, respectively, where 23.83% of them are heuristic-prone and 76.17% of them are non-heuristic-prone in the training set, while 25.57% of them are heuristic-prone and 74.43% of them are non-heuristic-prone in the validation set.

4.4 The Proposed Features

Based on our statistical analysis in Section 3.4, we further analyze the training samples in each category, and propose five features to facilitate the representation of internal relation patterns

between CCI samples and their model proneness, thereby assisting category prediction. The details of the five features are introduced as follows.

- **ReplaceRate:** Ignoring the code edits whose s_i^{cd} and $s_i^{\prime cd}$ are both meaningless symbols, i.e., ".", "*<con>*" and "∅", it is the ratio of code edits with the "replace" action among those with non-equal actions in a code change sequence:

$$ReplaceRate = \frac{n_{replace}}{n_{replace} + n_{insert} + n_{delete}}, \tag{8}$$

  where $n_{replace}$, n_{insert}, and n_{delete} denote the number of code edits with the action of "replace", "insert", and "delete" in a code change sequence, respectively. The intuition is that the code change sequence (as shown in Figure 2) that contains more code edits with the "replace" action in proportion is more likely to be the code-indicative changes, and tends to be easily and correctly synchronized by HebCUP.
- **MatchedLevelsNum:** This feature counts the number of matching levels of HebCUP that are satisfied by a CCI sample. The HebCUP totally designed three matching levels, i.e., tokens, sub-tokens, and sub-tokens without *<con>*. Therefore, the range of this feature is 0-3. The fact behind it is that if none of the matching levels is reached, HebCUP cannot conduct the substitution with its prepared replacement pairs, thus, the old comment will not be synchronized at all. On the contrary, if there are more levels of matchings found, HebCUP will have more chances to conduct the *correct synchronization* for old comments. More formally, we define the MatchedLevelsNum as below:

$$MatchedLevelsNum = isMatch_{(1)} + isMatch_{(2)} + isMatch_{(3)}, \tag{9}$$

  where $isMatch_{(i)}$ denotes whether the matching algorithm designed by HebCUP at the level i is satisfied. If it is satisfied, return 1; otherwise, return 0.
- **NonLetterCount:** Ignoring the meaningless symbols in code, i.e., ".", "*<con>*" and "∅", it counts the number of non-letter sub-tokens[3] in the union set of respective distinctive sub-token sets of the old and new code sequence. The intuition of this feature is that the more non-letter sub-tokens in the changes between old and new code, the more difficulties for HebCUP to sense the correct correspondences between code changes and comment changes. More formally, the NonLetterCount can be defined as below:

$$NonLetterCount = Count_{NonLetterToken}((S^{\prime cd} - S^{cd}) \cup (S^{cd} - S^{\prime cd})) \tag{10}$$

  where $S^{\prime cd}$ and S^{cd} denote the sets of sub-tokens in a new code sequence and an old code sequence, respectively; $Count_{NonLetterToken}$ denotes the function of counting for non-letter sub-tokens in a given set.
- **LongestChangedSeq:** It is the length of the longest continuous changed sub-sequence in the code change sequence after ignoring the code edits whose s_i^{cd} and $s_i^{\prime cd}$ are both meaningless symbols, i.e., ".", "*<con>*" and "∅". The motivation of designing this feature is that HebCUP has difficulty in finding the relationship between code changes and comment changes when the length of the continuously changed code sub-sequence is long [50].
- **TotalChangedNum:** Ignoring the code edits whose s_i^{cd} and $s_i^{\prime cd}$ are both meaningless symbols, i.e., ".", "*<con>*" and "∅", it is the total number of code edits with non-equal actions in a code change sequence:

$$TotalChangedNum = n_{replace} + n_{insert} + n_{delete}. \tag{11}$$

[3]A non-letter sub-token refers to the sub-token with characters apart from a-z and A-Z, such as "link1" and "&&".

Table 4. The Feature Values of the CCI Sample Shown in Listing 1

ReplaceRate	MatchedLevelsNum	NonLetterCount	LongestChangedSeq	TotalChangedNum
0.5	3	0	2	2

The principle behind this feature is that the more changed sub-tokens exist in the code change sequence, the more difficulty for HebCUP to correctly replace the sub-tokens in old comments when synchronizing [50].

To interpret our proposed features more clearly, we illustrate the CCI sample shown in Listing 1 as an example and list each feature value of the sample in Table 4. As can be seen, the ReplaceRate value of the CCI sample is 0.5, because it contains one code edit with the "replace" action among a total of two code edits with non-equal actions after ignoring the code edits whose s_i^{cd} and $s_i^{\prime cd}$ are both meaningless symbols. The MatchedLevelsNum value is 3, because HebCUP can find all matching levels of tokens, sub-tokens, and sub-tokens without *<con>* between code changes and the old comment according to its program running result. The NonLetterCount value is 0, because there is no non-letter sub-token in the code change sequence after ignoring the meaningless symbols. Both the LongestChangedSeq and TotalChangedNum values are 2, since the longest continuous changed sub-sequence and the total changed code edits are both {<"the","this","replace">, <"∅", "onos", "insert">} after ignoring the meaningless symbols.

Subsequently, we perform a series of statistical tests on the training set to further prove and quantify the correlation between those features and the class label (i.e., either heuristic-prone or non-heuristic-prone). Since the features we proposed are continuous variables and the class label is a binary variable, we employ the **Point-Biserial Correlation Coefficient (PBCC)**, a measurement suitable for evaluating the correlation between a binary variable and a continuous variable [48], to measure the degree of correlation between each feature and the class label. The larger the absolute value of the correlation coefficient, the larger the degree of correlation, and the sign of the correlation coefficient represents the positive or negative correlation. According to Table 5, we find that ReplaceRate is the most relevant feature to discriminate heuristic-prone and non-heuristic-prone samples, while TotalChangedNum is the least relevant feature, but still has a relatively great correlation with the class label. In addition, since the distribution of features in each category is unknown, we conduct a series of Mann-Whitney U tests [56] to evaluate whether the correlation between features and the class label is statistically significant. In addition, we also utilize the **Benjamini–Hochberg (BH)** procedure to adjust *p-value*s because we perform multiple independent tests, adjusting *p-value*s with BH helps us to control for the fact that sometimes small *p-value*s (less than 0.05) happen by chance, which could lead us to incorrectly reject the true null hypotheses [22]. As can be seen in Table 5, the last four features statistically significantly correlate with the label at the significance threshold of 0.05, 0.01, 0.005, and 0.001, and the first feature statistically significantly correlate with the label at the significance threshold of 0.05, 0.01, and 0.005. Figure 8 presents the training sample distribution of each category by violin plots among each proposed feature, where the white point represents the median value, and the black box indicates the interquartile range. Intuitively, heuristic-prone samples tend to have higher ReplaceRate and MatchedLevelsNum values, while non-heuristic-prone samples generally have higher NonLetterNum, LongestChangedSeq, and TotalChangedNum values. The distribution of samples indicates the intuitions behind our proposed features are correct.

4.5 Multi-Subsets Ensemble Learning

After labeling the samples and extracting the five features, the ith CCI sample for category prediction can be represented as $M_i = (\boldsymbol{x}_i, y_i)$, where $\boldsymbol{x}_i = (x_i^1, x_i^2, \ldots, x_i^5)$ is a 5-dimensional feature vector,

Table 5. The Description of Our Proposed Five Features

Features	**Description**	**PBCC (p-value)**
ReplaceRate	Ratio of code edits with the "replace" action among those with non-equal actions in a code change sequence.	0.3641 (2.51e-3)
MatchedLevelsNum	The number of matching levels of HebCUP that are satisfied by a CCI sample.	0.3510 (7.47e-5)
NonLetterCount	The number of non-letter sub-tokens in the union set of respective distinctive sub-token sets of the old and new code sequences.	−0.3150 (5.61e-8)
LongestChangedSeq	Length of the longest continuous changing sub-sequences between old and new code.	−0.2758 (1.14e-6)
TotalChangedNum	Total number of code edits with non-equal actions in a code change sequence.	−0.2617 (1.29e-5)

$^{\Phi}$PBCC is the abbreviation of Point-Biserial Correlation Coefficient.

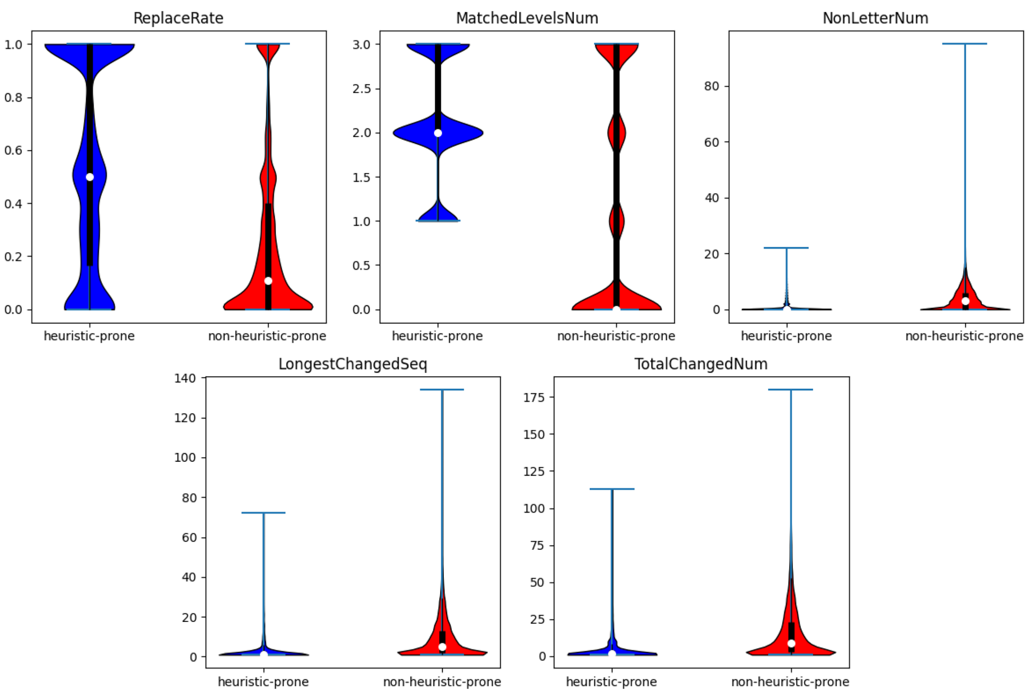

Fig. 8. The distributions of heuristic-prone and non-heuristic-prone samples among each proposed feature.

and y_i is its category. Then, the whole dataset of CCI samples can be represented as:

$$S = \{M_1, M_2, \ldots, M_n\} \tag{12}$$

where n is the number of samples in S. Since the dataset S contains more non-heuristic-prone samples (i.e., the majority class samples) than heuristic-prone ones (i.e., the minority class samples) as stated in the statistics of Section 4.3, the prediction model trained on the imbalanced dataset will focus more on the non-heuristic-prone samples, and are prone to predict the testing samples as non-heuristic-prone. However, accurately classifying the heuristic-prone samples is critical for boosting the performance of our proposed CBS approach, because the comments of more samples can be correctly synchronized by HebCUP. Inspired by Wu et al.'s study [90], we propose a Multi-Subsets Ensemble Learning (MSEL) algorithm to alleviate the class imbalance problem for category

prediction. More formally, the MSEL algorithm can be defined as below:

$$\hat{y}_i = F(\boldsymbol{x}_i) \tag{13}$$

where the $\hat{y}_i$ denotes the predicted category via the MSEL algorithm for the sample $\boldsymbol{x}_i$, F is the function describing the MSEL algorithm. Algorithm 1 lists the detailed process, and Figure 9 shows the overall framework of the MSEL algorithm, which consists of two stages with multiple steps.

The offline training stage includes the two steps, i.e., Balanced Multi-Subsets Construction and Ensemble Learning on Multi-Subsets.

- Balanced Multi-Subsets Construction: According to the ratio of the non-heuristic-prone samples to the heuristic-prone ones, we set the number of constructed subsets to be $K = \lfloor N_{major}/N_{minor} + 0.5 \rfloor$, where N_{major} is the number of the majority class samples (i.e., non-heuristic-prone samples) and N_{minor} is the number of the minority class samples (i.e., heuristic-prone samples) (Line 2). Then, we randomly choose N_{minor} non-heuristic-prone samples without replacement $K-1$ times to obtain $K-1$ chunks (Lines 4–5), and the remaining non-heuristic-prone samples are used as the last chunk (Line 7). Therefore, the K chunks are mutually exclusive, and each chunk, except for the last chunk,[4] contains the same number of non-heuristic-prone samples. Finally, we construct K balanced subsets by merging each chunk with all heuristic-prone samples (Line 9).
- Ensemble Learning on Multi-Subsets: After constructing multiple balanced subsets, we train a classifier h_k on each subset S_k (Line 12). Therefore, we can obtain multiple trained base classification models with heterogeneous decision boundaries.

For the online testing stage, with the input of the five features extracted from the testing sample M_{test}, each base classification model returns its predicted category, which counts as one vote (Lines 15–17). Finally, the MSEL algorithm counts the votes and assigns the category with the most votes to M_{test} (Line 18), thereby classifying for M_{test}.

5 EXPERIMENTS FOR CATEGORY PREDICTION

This section describes the experiments for category prediction, including the experimental setup, evaluation metrics, and experimental results.

5.1 Experimental Setup

Since there is no previous study on the CCI sample category definition and classification, here we adopt some widely used machine learning classifiers to plug in the MSEL algorithm as baselines in this field. Specifically, we investigate the performance of the five machine learning classifiers,[5] i.e., Decision Tree [16], Random Forest [15], LightGBM [39], Naive Bayes [70], and **Multi-Layer Perceptron (MLP)** [28] in this article, where their inputs are our proposed features introduced in Section 4.4. Our adoption of these classifiers is inspired by a similar task of comment classification, in which they have been proved to be effective [17, 63, 64, 95]. Besides, we also adopt **Bi-directional Long Short-Term Memory (Bi-LSTM)** [32, 72] and **Convolutional Neural Network (CNN)** [11] as extra classifiers to extract the semantics of code changes and old comments directly in the category classification for CCI samples.[6]

[4]If the last chunk accounts for less than half the size of other chunks, it merges into the second last chunk, thus the last chunk will be larger than other chunks; otherwise, we remain it intact, and the last chunk will be smaller compared with other chunks.

[5]We adopt sci-kit learn toolkit [65] to implement the Decision Tree, Random Forest, Naive Bayes, and Multi-Layer Perceptron. We adopt the official library [39] to implement LightGBM.

[6]We adopt TensorFlow [8] to implement the Bi-LSTM and CNN classifiers.

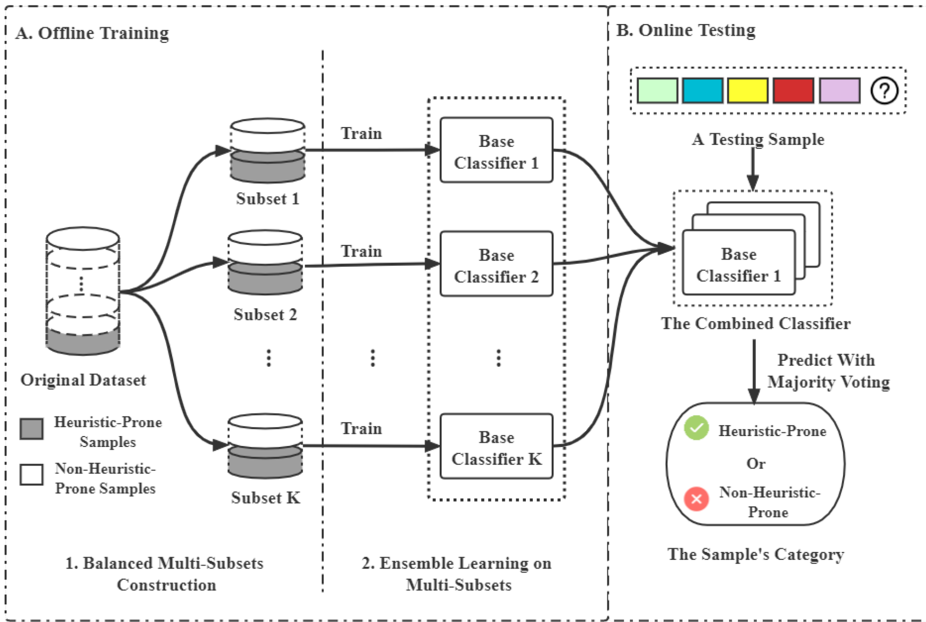

Fig. 9. The overview of the Multi-Subsets Ensemble Learning (MSEL) algorithm.

More specifically, for the construction of the Bi-LSTM classifier, we use four embedding layers to vectorize old code, new code, edit actions, and old comments, respectively. Then, the embedded old code, new code, and edit actions of each sample are concatenated horizontally for alignment, such that we can capture the code change semantics. Next, the Bi-LSTM network accepts the embeddings of each pair of code change and its associated old comment, and inputs them into their respective layers with two concatenated LSTMs (i.e., Bi-LSTM) for each. After horizontally propagating the semantics of code change and old comment from the two directions, respectively, their embeddings are concatenated together and go through a softmax layer to obtain the output labels. Whereas for the construction of the CNN classifier, we replace the Bi-LSTM structures with 1-dimensional CNN structures in the Bi-LSTM classifier, and keep other structures with no change.

Given the labeled training and validation set as described in Section 4.3, we merge them together as a mixed dataset, and employ the 10-fold cross-validation on it to evaluate each base classifier embedded in the MSEL algorithm in terms of the metrics described in Section 5.2. The 10-fold cross-validation divides the mixed dataset into 10 consecutive folds. Each fold (i.e., 8,941 samples) is used once for evaluation while the remaining nine folds (i.e., 80,476 samples) are used for training, thereby ensuring each fold is used as both the training and validation data. For the deep learning classifiers in particular, we train the models with 50 epochs and randomly select 10% of data for in-training-validation in each fold. Besides, we adopt an early stopping strategy that if the loss of in-training-validation stops decreasing for three epochs, the model will stop training.

Since hyper-parameter tuning is essential in machine learning and deep learning experiments, we generate candidated hyper-parameter settings (see Table 6 for details) according to a series of relevant hyper-parameter optimization studies [27, 37, 42, 81, 94], and control the capacity of them based on a given budget threshold (i.e., the tuneLength) for evaluation. The budget threshold refers to the number of different values to be evaluated for each hyper-parameter. As suggested by Kuhn [46], we use a budget threshold of 5. For each 10-fold cross-validation, we can only obtain an evaluation result of one classifier with one hyper-parameter combination. As such, to explore the

ALGORITHM 1: The MSEL Algorithm

Input: (1) The training set S_{train} containing N_{minor} heuristic-prone samples and N_{major} non-heuristic-prone samples; (2) A testing sample M_{test}.
Output: The predicted category y_{test} of the testing sample M_{test}.

1: /*offline training stage*/
2: Compute the number of subsets $K = \lfloor N_{major}/N_{minor} + 0.5 \rfloor$;
3: **for** $k = 1$ to K **do**
4: **if** $k < K$ **then**
5: $chunk_k$ = the randomly chosen N_{minor} samples from the non-heuristic-prone samples without replacement;
6: **else**
7: $chunk_k$ = the rest of non-heuristic-prone samples;
8: **end if**
9: Construct the kth subset S_k by merging $chunk_k$ and all heuristic-prone samples;
10: **end for**
11: **for** $k = 1$ to K **do**
12: Train the base classifier h_k on the subset S_k;
13: **end for**
14: /*online testing stage*/
15: **for** $k = 1$ to K **do**
16: y_{test}^k = The predicted category of M_{test} by the base classifier h_k;
17: **end for**
18: Majority voting among $\{y_{test}^1, y_{test}^2, \ldots, y_{test}^K\}$ to get the final prediction result of M_{test};
19: **return** The predicted category of M_{test};

Table 6. Hyper-Parameter Tuning for Category Prediction

Classifier	Hyper-parameter Name	Hyper-parameter Description	Candidated Values
Naive Bayes	var_smoothing	For calculation stability.	{0, 1e-9, 1e-7, 1e-5, 1e-3}
Random Forest	max_depth	The maximum depth of the tree.	{7, 9, 11, 13, 15}
	n_estimators	The number of trees in the forest.	{50, 100, 150, 200, 250}
Decision Tree	max_depth	The maximum depth of the tree.	{7, 9, 11, 13, 15}
LightGBM	max_depth	The maximum depth of the tree.	{7, 9, 11, 13, 15}
	n_estimators	The number of trees in the forest.	{50, 100, 150, 200, 250}
	learning_rate	The boosting learning rate.	{0.001, 0.005, 0.01, 0.05, 0.1}
MLP	hidden_layer_sizes	The number of neurons in the hidden layer.	{32, 64, 128, 256, 512}
	learning_rate_init	The initial learning rate used.	{0.001, 0.005, 0.01, 0.05, 0.1}
Bi-LSTM	units	Dimensionality of the output space.	{32, 64, 128, 256, 512}
	learning_rate	The initial learning rate used.	{0.001, 0.005, 0.01, 0.05, 0.1}
CNN	kernel_size	The length of the 1D convolution window	{1, 3, 5, 7, 9}
	filters	Dimensionality of the output space.	{32, 64, 128, 256, 512}
	learning_rate	The initial learning rate used.	{0.001, 0.005, 0.01, 0.05, 0.1}

best hyper-parameter combination of each classifier and compare them with each other, we utilize the grid search [14] such that we can conduct an exhaustive search through our manually specified set of the hyper-parameter space [81]. The tuned classifier that obtains the best performance is selected and plugged into the MSEL algorithm of CBS for online testing.

Table 7. The Evaluation Metrics for Category Prediction

	Actual	Predicted	
		heuristic-prone	non-heuristic-prone
	heuristic-prone	TP	FP
	non-heuristic-prone	FN	TN
Precision	TP/(TP+FP)		
Recall	TP/(TP+FN)		
$F1 - Score$	$2 \times$ Precision $\times$ Recall/(Precision + Recall)		

5.2 Evaluation Metrics for Category Prediction

We employ Precision, Recall, and F1-Score to evaluate the effectiveness of the MSEL algorithm with different base classifiers, and Table 7 lists the definitions of the metrics. In our experiment, the heuristic-prone samples are regarded as positive, and the non-heuristic-prone samples are regarded as negative. In Table 7, **TP (True Positive)** is the number of heuristic-prone samples that are correctly predicted to be heuristic-prone, **FP (False Positive)** is the number of non-heuristic-prone samples that are wrongly predicted to be heuristic-prone, **FN (False Negative)** is the number of heuristic-prone samples that are wrongly predicted to be non-heuristic-prone, and **TN (True Negative)** is the number of non-heuristic-prone samples that are correctly predicted to be non-heuristic-prone.

Precision is the ratio of the correctly predicted heuristic-prone samples to all predicted heuristic-prone samples.

Recall is the ratio of the correctly predicted heuristic-prone samples to all actual heuristic-prone samples.

F1-Score is the harmonic mean of Precision and Recall.

Since the proportion of categories in each fold may vary more or less, we aggregate the TP/FP/TN/FN instances across the 10 folds and then compute precision, recall, and F1-Score, rather than computing these three metrics for each fold and then averaging them, thereby obtaining a more accurate evaluation for each classifier.

5.3 Experimental Results

Table 8 presents the performance of the MSEL algorithm with different base classifiers, where the hyper-parameters of these classifiers are fully tuned before conducting the comparison. The best hyper-parameter combinations are listed in the last column. According to our observation, we have the following findings:

(1) LightGBM performs best in terms of F1-Score. Besides, it also ranks second in terms of Recall and ranks third in terms of Precision. This indicates that MSEL with LightGBM can reach the best balance between increasing the true positive rate and decreasing the false positive rate.

(2) Naive Bayes performs the best in terms of Recall. However, its performance on Precision and F1-Score are the lowest among seven classifiers. It indicates that the MSEL algorithm with the Naive Bayes as the base classification model tends to predict the samples to be heuristic-prone, which can recognize more heuristic-prone samples but with lots of false positives.

(3) CNN obtains the best performance on Precision, and the second best performance on F1-Score. Nevertheless, it does not perform well on Recall comparing with other classifiers, where it ranks sixth. This indicates that MSEL with CNN has the highest true positive rate but can classify out relatively small number of true heuristic-prone samples.

(4) F1-Score is the harmonic mean of Precision and Recall, which is a more comprehensive metric for the evaluation of category prediction for CCI samples. Therefore, although Naive Bayes

Table 8. The Performance of the MSEL Algorithm with Different Base Classifiers Across the 10-fold Cross-validation

Classifier	Precision	Recall	F1-Score	Best Hyper-parameter Combination
Naive Bayes	47.69%	**91.07%**	62.60%	{“var_smoothing”: 1e-9}
Random Forest	59.36%	88.89%	71.18%	{“max_depth”: 13, “n_estimators”: 200}
Decision Tree	59.46%	88.68%	71.19%	{“max_depth”: 15}
LightGBM	59.69%	90.57%	**71.96%**	{“max_depth”: 11, “n_estimators”: 250, “learning_rate”:0.1}
MLP	56.77%	87.81%	68.96%	{“hidden_layer_sizes”: 512, “learning_rate”:0.01}
Bi-LSTM	60.20%	85.36%	70.60%	{“units”: 128, “learning_rate”:0.01}
CNN	**61.00%**	86.88%	71.68%	{“filters”: 32, “kernel_size”:3, “learning_rate”:0.01}

performs better in terms of Recall, and CNN performs better in terms of Precision, LightGBM is selected as the base classifier of the MSEL algorithm for its best performance in terms of F1-Score.

Conclusion

LightGBM embedded in the MSEL algorithm performs the best on the category prediction for CCI samples. Thus, LightGBM is selected as the base classifier of the MSEL algorithm in our CBS approach.

6 EXPERIMENTS FOR CODE-COMMENT SYNCHRONIZATION

In this section, we illustrate the experiments of code-comment synchronization, which contains the experimental setup, evaluation metrics, and experimental results.

6.1 Experimental Setup

After evaluating and selecting the best base classifier, we re-train the MSEL algorithm on the whole mixed dataset. Then the trained MSEL algorithm is applied to the testing set to conduct the category prediction for CCI samples. Finally, with the HebCUP and trained CUP[7], the CBS approach can be constructed as a whole. In this section, we compare the performance of CBS, CUP, and HebCUP on the testing set in terms of Accuracy, Recall@5, AED, RED, BLEU-4, and ESS ratio. As such, we can explore whether the category prediction can further boost the code-comment synchronization performance via the comprehensive assessment. Subsequently, we design a CBS-R approach with random classification for all CCI samples, and make a comparison with CBS to verify whether the effect of the category prediction model is significant. Finally, we further devise a CBS-Max approach with completely correct classification for all CCI samples, and make a comparison with CBS to report the maximum capability of the CBS approach.

6.2 Evaluation Metrics

As we mentioned above, we utilize Accuracy, Recall@5, AED, RED, BLEU-4, and ESS ratio as our evaluation metrics to assess the performance of each model. The first five metrics are utilized by the previous studies as we illustrated in the Section 3.2. The last metric, ESS ratio, is first proposed in this paper to evaluate the ratio of samples whose edit distances are reduced after the code-comment synchronization. These samples can be referred to as Effectively Synchronized Samples (ESS) because their synchronized comments are closer to the reference comments, which can be regarded as effective synchronizations. This metric measures the ratio of samples (i.e., code methods) that code-comment synchronizers can relieve the effort of developers, the higher, the better. The difference between ESS Ratio and AED as well as RED is the measuring granularity, where

[7]Here we re-train the CUP with the same hyper-parameters and training strategies as recorded in its original paper [53].

the former is on the sample-level while the latter two are on the sub-token-level. As the complementary metrics of AED and RED, ESS ratio can be formally defined as below, where N denotes the total number of the samples, $N_{ED_diff<0}$ denotes the number of samples whose edit distances are reduced.

$$ESS\ ratio = \frac{N_{ED_diff<0}}{N} \tag{14}$$

Considering the existence of CUP in our proposed CBS approach, we run the experiment ten times and report confidence interval ($\mu \pm \theta$) at the confidence level of 95% for each of the metrics,[8] because training for CUP has the randomness characteristic due to its deep learning-based structure, and we also hope to obtain a stable result. More formally, we show the computation of μ and θ as below, where μ is the sample mean, θ is the range around μ that we estimated, $t_{0.025}(N_e - 1)$ is the two-tails t distribution table value with the degrees of freedom of $N_e - 1$ at the confidence level of 95%,[9] s is the sample standard deviation, N_e is the number of experiments, and v_i represents the value of Accuracy, Recall@5, AED, RED, BLEU-4, or ESS ratio in the ith experiment.

$$\mu = \frac{1}{N_e}\sum_{i=1}^{N_e} v_i, \tag{15}$$

$$\theta = \frac{s}{\sqrt{N_e}} t_{0.025}(N_e - 1), \tag{16}$$

$$s = \sqrt{\frac{1}{N_e - 1}\sum_{i=1}^{N_e}(v_i - \mu)}, \tag{17}$$

6.3 Experimental Results

Visualizations: In Table 9, we list a detailed description for the performance of CBS. The rows of HebCUP Side and CUP Side demonstrate their respective code-comment synchronization performance on their assigned testing samples (i.e., samples predicted as heuristic-prone are piped into the HebCUP side for code-comment synchronization while samples predicted as non-heuristic-prone are piped into the CUP side). After combining the performance of HebCUP Side and CUP Side, the final performance of CBS is displayed in the last row. Since we run the experiment ten times to eliminate the randomness of CBS and CUP, we report the confidence interval as we mentioned in Section 6.2 for each evaluation metric. However, HebCUP need not be reported in this way due to the heuristic characteristic. Table 10 presents the performance of CBS, CUP, and HebCUP in terms of Accuracy, Recall@5, AED, RED, BLEU-4, and ESS ratio on the testing set also via confidence intervals. In order to investigate whether the superiority of our proposed CBS approach towards the two baselines is statistically significant, we conduct the Wilcoxon signed-rank test [88] on the experiment results for each evaluation metric. We adopt Wilcoxon signed-rank test because it is a paired difference test without the assumption that the paired samples are normally distributed. Similarly, our comparisons between each approach and CBS are paired in each experiment and their distributions are also unknown to us. In addition, we also utilize the Benjamini–Hochberg (BH) procedure to adjust *p-value*s for the same reason as we mentioned in Section 4.4 [22]. In Table 10, the performance value labeled with a star (*) indicates CBS statistically significantly outperforms the baseline in terms of the corresponding metrics. Besides, the row of CBS-R demonstrates the performance of CBS when the samples are randomly assigned, which is a lower

[8] The adoption of the confidence level at 95% is a follow of statistical convention [21, 44].

[9] We adopt t distribution to approximate the distribution of the repetitive experimental results of CBS because its population distribution is unknown [13].

Table 9. The Performance of Our Proposed CBS Approach on the Testing Set

Approach	Accuracy (%)	Recall@5 (%)	AED	RED	BLEU-4	ESS ratio (%)
HebCUP Side	59.51[2062.00/3465]	60.72	2.050	0.729	81.77	69.52
CUP Side	8.83 ± 0.34[506.80 ± 19.53/5739]	20.54 ± 0.40	4.474 ± 0.045	1.018 ± 0.010	67.96 ± 0.21	15.53 ± 1.14
CBS	27.91 ± 0.21[2568.80 ± 19.53/9204]	35.67 ± 0.25	3.561 ± 0.028	0.937 ± 0.007	73.16 ± 0.13	35.85 ± 0.71

$^{\Phi}$The number of *correct synchronizations* are presented in square brackets via the format of "[$\mu \pm \theta$/assigned testing samples]".

Table 10. The Performance of Our Proposed CBS Approach and Baselines on the Testing Set

Approach	Accuracy (%)	Recall@5 (%)	AED	RED	BLEU-4	ESS ratio (%)
HebCUP	25.71[2367.00]*	26.39*	3.742*	0.985*	71.96*	31.99*
CUP	20.16 ± 0.73[1855.90 ± 67.03]*	32.28 ± 0.74*	3.606 ± 0.050*	0.949 ± 0.013*	71.99 ± 0.24*	28.30 ± 1.21*
Ours						
CBS-R	22.79 ± 0.24[2094.40 ± 11.05]*	29.12 ± 0.22*	3.676 ± 0.021*	0.968 ± 0.005*	71.80 ± 0.01*	30.85 ± 0.87*
CBS	**27.91 ± 0.21[2568.80 ± 19.53]**	**35.67 ± 0.25**	**3.561 ± 0.028**	**0.937 ± 0.007**	**73.16 ± 0.13**	**35.85 ± 0.71**
CBS-Max	30.88 ± 0.09[2842 ± 8.55/9204]	38.97 ± 0.31	3.378 ± 0.031	0.889 ± 0.008	75.51 ± 0.14	37.93 ± 0.74
W/T/L						
- HebCUP	10/0/0	10/0/0	10/0/0	10/0/0	10/0/0	10/0/0
- CUP	10/0/0	10/0/0	10/0/0	10/0/0	10/0/0	10/0/0
- CBS-R	10/0/0	10/0/0	10/0/0	10/0/0	10/0/0	10/0/0

$^{\Phi}$The number of *correct synchronizations* are presented in square brackets via the format of "[$\mu \pm \theta$]", where the number of total testing samples is 9,204. Besides, *refers to the CBS outperforms the corresponding approach statistically significantly and their *p-value*s all equals to 1.95e-3 < 0.005.

limit customized by ourselves. And the row of CBS-Max presents the best performance that CBS can reach when the categories of all samples can be correctly predicted. The rows of -HebCUP, -CUP, -CBS-R under the **W/T/L (Win/Tie/Loss)** record how many experiments CBS obtains a win, tie, and loss compared to CUP, HebCUP, and CBS-R in terms of each evaluation metric.

Results: We have the following findings from the Tables 9 and 10:

(1) According to Table 9, 3,465 (37.65%) samples are predicted to be heuristic-prone while 5,739 (62.35%) samples are predicted to be non-heuristic-prone by the MSEL algorithm among the whole testing set of 9,204 samples in total. On the basis of the sample allocation results, HebCUP Side achieves an average score of 59.51% in terms of Accuracy, 60.72% in terms of Recall@5, 2.050 in terms of AED, 0.729 in terms of RED, 81.77 in terms of BLEU-4, and 69.52% in terms of ESS ratio on the 3,465 predicted heuristic-prone samples, while CUP Side achieves an average score of 8.83%, 20.54%, 4.474, 1.018, 67.96, and 15.53% in terms of Accuracy, Recall@5, AED, RED, BLEU-4, and ESS ratio on the 5,739 predicted non-heuristic-prone samples. Combining the above performance of the two models, CBS finally achieves an average score of 27.91% in terms of Accuracy, 35.67% in terms of Recall@5, 3.561 in terms of AED, 0.937 in terms of RED, 73.16 in terms of BLEU-4, and 35.85% in terms of ESS ratio.

(2) Similar to the experimental results on the validation set in Section 3.4, CUP still outperforms HebCUP in terms of Recall@5, AED, RED, and BLEU-4, whereas HebCUP has its superiority in terms of Accuracy and ESS ratio on the testing set as shown in Table 10.

(3) Comparing with the two baselines, CBS performs the best in terms of all evaluation metrics as shown in Table 10. Specifically, CBS outperforms HebCUP by 8.53% in terms of Accuracy, 35.16% in terms of Recall@5, 4.85% in terms of AED and RED, 1.67% in terms of BLEU-4, and 12.09% in terms of ESS ratio. CBS also outperforms CUP by 38.41% in terms of Accuracy, 10.52% in terms of Recall@5, 1.24% in terms of AED and RED, 1.62% in terms of BLEU-4, and 26.69% in terms of ESS ratio.

(4) The row of CBS-R in Table 10 shows the performance of CBS with random category classification. We find that CBS outperforms CBS-R by 22.48% in terms of Accuracy, 22.50% in terms of Recall@5, 3.12% in terms of AED and RED, 1.89% in terms of BLEU-4, and 16.20% in terms of

ESS ratio. This demonstrates that the accurate category prediction is substantially significant to CBS for code-comment synchronization. Besides, the row of CBS-Max in Table 10 shows the upper limit of CBS. As can be seen, the Accuracy, Recall@5, BLEU-4, and ESS ratio of CBS can reach to 90.35%, 91.53%, 96.89%, 94.53% of those of CBS-Max. Both AED and RED values of CBS can reach to 105.44% of those of CBS-Max.[10]

(5) The row of -HebCUP, -CUP, and -CBS-R under W/T/L in Table 10 shows that CBS outperforms HebCUP, CUP, and CBS-R ten times in terms of all evaluation metrics. The Wilcoxon signed-rank test also shows that CBS statistically significantly outperforms CUP, HebCUP, and CBS-R for all evaluation metrics at the significance threshold of 0.05, 0.01, and 0.005, where the *p-value*s for each are all equal to 1.95e-3.[11]

Conclusion

(1) CBS statistically significantly outperforms CUP and HebCUP in terms of all evaluation metrics, because of the classifying before synchronizing strategy.
(2) The accurate category prediction is substantially significant to CBS for code-comment synchronization.

7 FURTHER EXPLORATION FOR CBS

In this section, we propose a series of **Research Questions (RQs)** to further analyze the performance of CBS.

7.1 RQ1: How Do Other Classifiers Influence the Performance of CBS?

Motivation: Although we have selected LightGBM as the base classifier of the MSEL algorithm due to its achievement of the highest F1-Score under the 10-fold cross-validation, exploring the performance of CBS with other classifiers is also necessary. Thus, we also list their corresponding experimental results (reported by confidence intervals) in Table 12 to do further analysis. In addition, since the code-comment synchronization results depend heavily on the sample allocation guided by the MSEL algorithm, we also present Table 11 to demonstrate the category prediction results of the MSEL algorithm with different base classifiers on the testing set.

Results: According to Table 11, still, LightGBM embedded in MSEL performs better than other classifiers in terms of F1-Score on the testing set, Naive Bayes achieves the highest Recall value. Decision Tree substitutes CNN and performs best in terms of Precision. Except for the Bi-LSTM and CNN, the performance of the rest of the classifiers is almost consistent with those under the 10-fold cross-validation. It is worth noting that, comparing with the 10-fold cross-validation, the performance of Bi-LSTM and CNN drops a lot on the testing set. Thus, we infer that the patterns learned by Bi-LSTM and CNN directly from code changes and old comments do not match those in the test set, leading to the overfitting of these two models. The unmatched pattern learned may attribute to the difficulty in directly learning on the raw data, which is not intuitive enough to capture the differences between heuristic and non-heuristic samples, especially compared with our proposed features. On the contrary, other classifiers exploit our proposed features can intuitively

[10] The ratio here exceeds 100% because, as we mentioned in Section 3.2, for RED and AED, the smaller, the better. 105.44%=0.937/0.889 (i.e., the RED of CBS/the RED of CBS-Max, taking RED an example), which illustrates CBS-Max is better than CBS in terms of RED and AED.

[11] The *p-value*s are the same here because Wilcoxon signed-rank test only takes into account the signs of the differences of values of every pair of data, and it does not take into account how large is such a difference. In our comparison, CBS outperforms each of the other approaches on all evaluation metrics, and each experiment, thus the *p-value*s via Wilcoxon signed-rank test are the same.

Table 11. The Category Prediction Results of the MSEL Algorithm with Different Base Classifiers on the Testing Set

Classifier	Precision	Recall	F1-Score
Naive Bayes	49.78%	**89.17%**	63.89%
Random Forest	59.50%	86.07%	70.36%
Decision Tree	**59.87%**	85.77%	70.52%
LightGBM	59.51%	87.11%	**70.71%**
MLP	59.57%	83.79%	69.64%
Bi-LSTM	49.73%	68.85%	57.74%
CNN	53.24%	70.45%	60.65%

Table 12. The Code-comment Synchronization Performance of CBS with Different Base Classifiers on the Testing Set

Classifier	Accuracy (%)	Recall@5 (%)	AED	RED	BLEU-4	ESS Ratio (%)
Naive Bayes	27.57 ± 0.22[2537.20 ± 20.04]	34.65 ± 0.25	3.625 ± 0.028	0.954 ± 0.007	72.65 ± 0.14	35.62 ± 0.71
Random Forest	27.79 ± 0.24[2557.60 ± 22.16]	35.61 ± 0.28	3.563 ± 0.028	0.938 ± 0.007	73.19 ± 0.13	35.70 ± 0.73
Decision Tree	27.77 ± 0.21[2555.70 ± 19.15]	35.58 ± 0.27	3.580 ± 0.029	0.942 ± 0.008	73.01 ± 0.14	35.77 ± 0.72
LightGBM	**27.90 ± 0.21[2567.80 ± 19.53]**	**35.67 ± 0.25**	**3.562 ± 0.028**	**0.937 ± 0.007**	**73.16 ± 0.13**	**35.85 ± 0.71**
MLP	27.69 ± 0.23[2549.00 ± 21.50]	35.49 ± 0.26	3.564 ± 0.030	0.938 ± 0.008	73.15 ± 0.14	35.62 ± 0.72
Bi-LSTM	25.47 ± 0.24[2344.60 ± 22.27]	33.71 ± 0.27	3.567 ± 0.027	0.939 ± 0.007	72.79 ± 0.12	32.89 ± 0.72
CNN	25.70 ± 0.22[2366.10 ± 20.70]	34.05 ± 0.27	3.564 ± 0.031	0.938 ± 0.008	72.84 ± 0.13	32.77 ± 0.73

[Φ]The performance on each metric is recorded via confidence interval ($\mu \pm \theta$) as introduced in Section 6.2, and the number of *correct synchronization*s are presented in square brackets, where the number of total testing samples is 9,204.

learn the differences between heuristic and non-heuristic samples, where their learned patterns can also be easily generalized to the unseen data.

On the other hand, as shown in Table 12, using LightGBM as the base classifier of the MSEL algorithm makes CBS perform best in terms of all evaluation metrics. We can also observe that the performance of CBS is approximately positively related with the F1-Score of category prediction models, which further indicates that CBS choosing the classifier with the highest F1-Score to embed in MSEL is reasonable.

In addition, as shown in Tables 10 and 12, CBS with all base classifiers, except for Naive Bayes, Bi-LSTM, and CNN, outperforms HebCUP and CUP in terms of all evaluation metrics. Even for using Naive Bayes as the base classification model, CBS still performs better than CUP and HebCUP in terms of Accuracy, Recall@5, BLEU-4, and ESS ratio. Besides, although Bi-LSTM and CNN do not exploit our proposed features, adopting them in CBS still outperforms HebCUP and CUP on Recall@5, BLEU-4, AED, RED and ESS ratio. Above results indicate that as long as the classifier embedded in the MSEL algorithm has some ability in the category prediction for CCI samples, CBS is definitely able to further boost the performance of code-comment synchronization.

7.2 RQ2: How Does the Multi-Subset Ensemble Learning (MSEL) Influence the Performance of CBS?

Motivation: In this study, we propose a Multi-Subsets Ensemble Learning (MSEL) algorithm to alleviate the class imbalance problem in the category prediction for CCI samples. The class imbalance problem would result in lots of actual heuristic-prone samples being predicted to be non-heuristic-prone due to their low percentage (around 23%-25% shown in Table 3) in the overall dataset, thus causing a great loss on the accurate code-comment synchronization of our CBS approach. Therefore, we validate the effectiveness of the MSEL algorithm in this subsection.

Table 13. The Experimental Results for Exploring the Effect of MSEL Algorithm

(A) The Category Prediction Result with/without MSEL Algorithm			
Structure	Precision	Recall	F1-Score
The individual LightGBM	**74.05%(1527/2062)**	64.51%(1527/2367)	68.95%
MSEL With LightGBM	59.51%(2062/3465)	**87.11%(2062/2367)**	**70.71%**

(B) The Code-Comment Synchronization Results of CBS with/without MSEL Algorithm						
Structure	Accuracy (%)	Recall@5 (%)	AED	RED	BLEU-4	ESS Ratio (%)
CBS without MSEL	25.49 ± 0.25[2346.1 ± 22.89]	34.55 ± 0.39	**3.510 ± 0.035**	**0.924 ± 0.009**	**73.38 ± 0.16**	33.26 ± 0.83
CBS with MSEL	**27.90 ± 0.21[2567.80 ± 19.53]**	**35.67 ± 0.25**	3.562 ± 0.028	0.937 ± 0.007	73.16 ± 0.13	**35.85 ± 0.71**

$^{\Phi}$The performance on each metric in (B) is recorded via confidence interval ($\mu \pm \theta$) as introduced in Section 6.2, and the number of *correct synchronization*s are presented in square brackets, where the number of total testing samples is 9,204.

Results: Table 13(a) presents the category prediction results of MSEL with LightGBM and the individual LightGBM algorithm on the testing set. As can be seen, using the multi-subsets ensemble learning strategy can largely improve the Recall value (up to 87.11%), which indicates that many more actual heuristic-prone samples (35.03% higher than using the individual LightGBM algorithm) can be found and assigned to HebCUP for *correct synchronization.* Although the Precision value decreases by 19.64%, the values of Recall and F1-score increase by 35.03% and 2.55%, respectively. It indicates that using the MSEL algorithm contributes to the overall performance improvement of category prediction for CCI samples.

Table 13(b) presents the code-comment synchronization results (reported by confidence intervals) of CBS with and without MSEL on the testing set. Despite the certain loss in terms of AED, RED (losses by 1.46%), and BLEU-4 (losses by 0.30%), CBS with MSEL outperforms that without MSEL in terms of Accuracy (improves by 9.49%), Recall@5 (improves by 3.23%), and ESS ratio (improves by 7.79%), showing that utilizing the MSEL algorithm can largely improve the success rate of *correct synchronizations* within both the first and five attempts. Simultaneously, the improvement on ESS ratio demonstrates the synchronized comments of more CCI samples become closer to their references. The difference of performance between CBS with and without MSEL originates from the sample allocation guided by the category prediction model. More specifically, despite the high precision, the CBS approach without MSEL finds fewer actual heuristic-prone samples (1,527 of 2,367 in total), leading to most of samples being assigned to and handled by CUP Side. As such, CUP Side dominates the CBS framework and exhibits its advantages of the generalization capability on the large proportion of the total samples, thereby achieving better performance in terms of AED, RED, and BLEU-4. However, since HebCUP Side only conducts the synchronization on a small part of samples, it contributes little to Accuracy, which further declines its contribution on Recall@5. Similarly, the ESS ratio drops due to the same reason. Thus, the CBS approach without MSEL does not perform well on these three metrics. On the contrary, although with relatively low precision, the CBS approach with MSEL identifies most of the actual heuristic-prone samples (2,062 of 2,367 in total), thus causing an opposite result compared with the CBS without MSEL.

In addition, we find that CBS without MSEL performs worse than HebCUP by 0.89% in terms of Accuracy, while CBS with MSEL statistically significantly outperforms CUP and HebCUP in terms of all evaluation metrics as we mentioned in Section 6.3. We further conduct the Wilcoxon signed-rank test [88] with Benjamini–Hochberg (BH) for *p-value* adjustment [22] to analyze whether there is a statistically significant difference between CBS without MSEL and the two baselines. The results show that the performance superiority of HebCUP against CBS without MSEL in terms of Accuracy is not statistically significant, where the *p-value* is 0.084 > 0.05. Yet, CBS without MSEL statistically significantly outperforms CUP and HebCUP in terms of all other metrics, where the *p-value*s are all equal to 1.95e-3 < 0.05. Since both approaches achieve better performance compared with the state-of-the-art baselines, we recommend practitioners to adopt the MSEL algorithm in

Table 14. Top Performing Feature Set of Each Level Evaluated Via 10-fold Cross-validation

Level	Feature Set	Precision	Recall	F1-Score
1	MatchedLevelsNum	40.73%	100%	57.89%
2	MatchedLevelsNum; TotalChangedNum	55.82%	89.46%	68.75%
3	ReplaceRate; MatchedLevelsNum; TotalChangedNum	58.14%	90.02%	70.65%
4	ReplaceRate; MatchedLevelsNum; LongestChangedSeq; TotalChangedNum	59.21%	90.21%	71.49%
5	ReplaceRate; MatchedLevelsNum; NonLetterCount; LongestChangedSeq; TotalChangedNum	59.69%	90.57%	71.96%

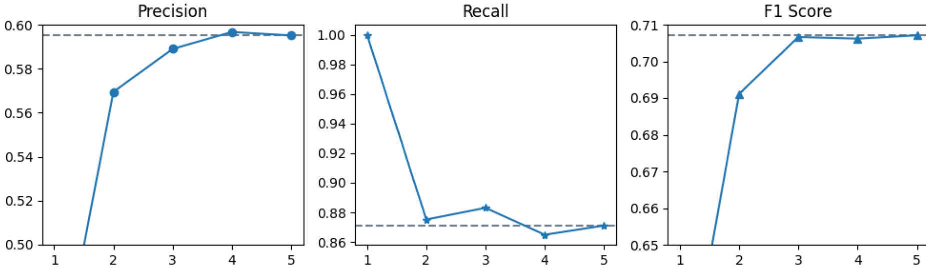

Fig. 10. The category prediction results on each evaluation metric among different feature levels.

CBS approach when they prefer more correct and effective synchronizations. On the contrary, if they prefer the CBS approach to be more generalized on CCI samples, they had better adopt the MSEL-free version of our CBS approach.

7.3 RQ3: How Do the Proposed Features Influence the Performance of CBS?

Motivation: We propose five original features, which we adopt in the MSEL algorithm with lightGBM, and obtain the best category prediction performance. Nevertheless, since different features have their heterogeneous characteristics of different degrees, and different combinations among these features will produce various performance on category prediction and further affect the code-comment synchronization results. Therefore, we investigate the impact of our proposed five features on CCI sample category prediction and code-comment synchronization.

Results: There are in total 31(= $C_5^1 + C_5^2 + C_5^3 + C_5^4 + C_5^5$) different feature combinations. For each one, we first conduct the 10-fold cross-validation experiment on the mixed dataset (i.e., training set + validation set) to evaluate its performance on the CCI sample category prediction. To simplify the experimental analysis, we partition the 31 feature combinations to five levels of feature sets (i.e., from *level*-1 with one feature to *level*-5 with five features). Next, the best performing (evaluated by F1-Score) feature combination among each level is selected for further comparison: {*level*-1: *MatchedLevelsNum*; *level*-2: *MatchedLevelsNum*, *TotalChangedNum*; *level*-3: *ReplaceRate*, *MatchedLevelsNum*, *TotalChangedNum*; *level*-4: *ReplaceRate*, *MatchedLevelsNum*, *LongestChangedSeq*, *TotalChangedNum*; *level*-5: *ReplaceRate*, *MatchedLevelsNum*, *NonLetterCount*, *LongestChangedSeq*, *TotalChangedNum*}, as shown in Table 14. This step cannot be conducted directly on the testing set because before the real comparison among the five groups, we hope to keep the testing set unseen to us.

Hereon, we narrow down the comparisons between feature combinations from 31 groups to 5 groups. Then, we adopt each selected feature combination of each level to experiment on the testing set for CCI sample category prediction, and present Figure 10 to display the variation of category prediction results with different levels of feature sets. We can find that CBS on feature

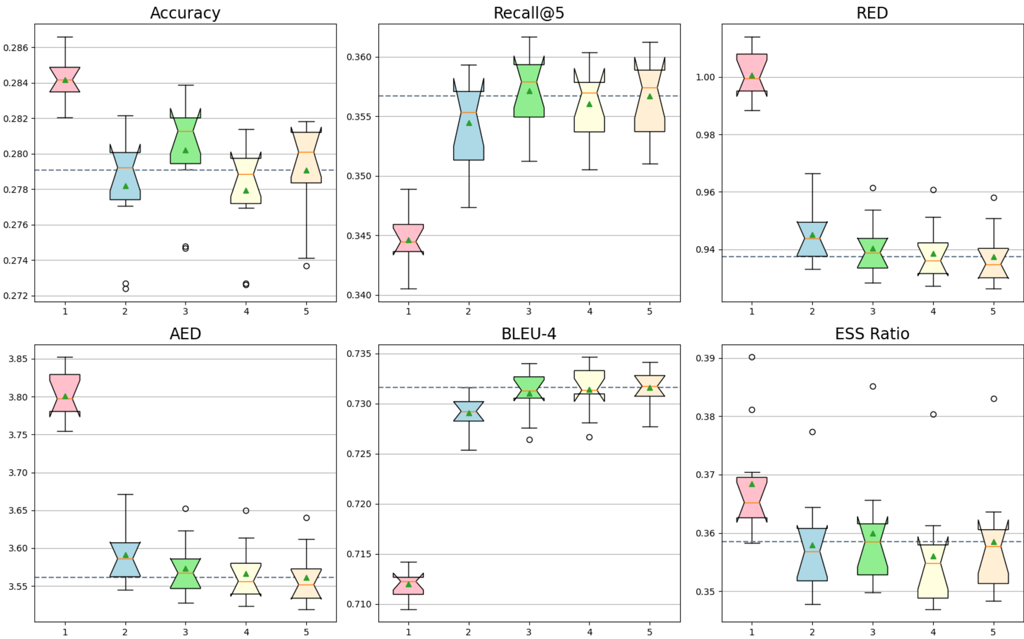

Fig. 11. The code-comment synchronization performance on each evaluation metric among different feature levels.

set *level*-4 performs the best in terms of Precision (59.67%), CBS on feature set *level*-1 performs the best in terms of Recall (100%), and CBS on feature set *level*-5 performs best in terms of F1-Score (70.71%). The feature set *level*-1 only includes *MatchedLevelsNum*, and CBS trained on this feature can recognize all actual heuristic-prone samples, thus achieves the score of 100% on Recall, but with very low Precision as introduced in Section 4.4. Afterward, with considering more and more features, decision boundaries of classification gradually become rigorous, leading to an increasing trend of the Precision and F1-Score.

Finally, we present the code-comment synchronization results of CBS with different feature combinations on the testing set in Figure 11. We find that (1) CBS with the feature set *level*-1 achieves the best performance in terms of Accuracy (28.42%) and ESS ratio (36.84%) on average but performs the worst in other metrics. Due to its extremely unbalanced performance on different evaluation metrics, we do not recommend adopting only one feature for category prediction in our CBS approach; (2) CBS with the feature set *level*-3 performs the best in terms of Recall@5 (35.72%) on average, and ranks the second in terms of Accuracy (28.07%) and ESS ratio (36.08%). Besides, it ranks the third on RED (0.941), AED (3.574), and BLEU-4 (73.08). (3) CBS with the feature set *level*-5, on average, performs the best in terms of RED (0.937), AED (3.562), and BLEU-4 (73.16), ranks the second in terms of Recall@5 (35.67%), and ranks the third in terms of Accuracy (27.90%) and ESS ratio (35.85%). The different superiority between CBS with the feature sets *level*-3 and *level*-5 is due to their different performance in the category prediction for CCI samples. Firstly, both feature sets achieve quite high performance on F1-Score, ranking second (70.66%) and best (70.71%), respectively. Besides, if we do not consider the feature set *level*-1, CBS with the feature set *level*-3 achieves the highest performance on Recall (88.31%), which causes relatively more actual heuristic-prone samples to be allocated to HebCUP Side. Therefore, in the subsequent code-comment synchronization, CBS with the feature set *level*-3 achieves higher performance in

Table 15. The Examples where CBS Fails

ID	Code Change	Comments

1	Project: plutext/docx4j
	Commit ID: 3151cd20c677418e5c5f4f0a3b9bc05ce7fc3611
	- public String getValue() {
	+ public AppearanceType getAppearance() {
	- return value;
	+ return appearance;
	}
		Old: Gets the value of the value property.
		New: Gets the value of the appearance property.
		HebCUP: Gets the Appearance of the Appearance property.
		CUP: Gets the value of the appearance property.
		CBS: Gets the Appearance of the Appearance property.

2	Project: OpenGamma/Strata
	Commit ID: 09358e034d05edb45dd6872234b56964d1b46701
	- public double getPrice() {
	+ public Optional<TradedPrice> getTradedPrice() {
	- return price;
	+ return Optional.ofNullable(tradedPrice);
	}
		Old Comment: Gets the price that was traded, in decimal form.
		New Comment: Gets the price that was traded, together with the trade date, optional.
		HebCUP: Gets the Optional.of Nullable that was traded, in decimal form.
		CUP: Gets the price that was traded, in decimal form.
		CBS: Gets the price that was traded,in decimal form.

terms of Accuracy and ESS ratio, where a large amount of correctly synchronized samples further contributes to Recall@5 and makes it higher. CBS with the feature set *level*-5 performs better in terms of Precision (59.51%) but with lower Recall (87.11%), compared with feature set *level*-3. Therefore, relatively more samples are allocated to CUP Side for code-comment synchronization, and the higher RED, AED, and BLEU-4 are achieved. However, since HebCUP Side only synchronizes a small part of samples in this case, it contributes little to ESS ratio and Accuracy, which further declines its contribution to Recall@5.

In summary, we recommend practitioners to select appropriate features according to their preferences when utilizing our CBS approach on the code-comment synchronization. If they prefer more correct and effective synchronizations, they can adopt {*ReplaceRate*, *MatchedLevelsNum*, *TotalChnagedNum*} to train the category classification model in CBS; while if they prefer the approach with a better generalization capability, they had better choose all five features.

8 DISCUSSION

This section discusses the situations CBS may fail and implications of improving code-comment synchronization via category prediction of CCI samples.

8.1 Where Does CBS Fail

Based on our analysis on the experimental results of CBS, we find it fails in two situations, i.e., (1) misclassification, and (2) even correctly classified, CUP side may not be able to conduct the *correct synchronization*. We present Table 15 with two samples to illustrate the above situations, where tokens to be deleted in old code or old comments are highlighted in red, tokens to be added or correctly updated are highlighted in green, and tokens that are wrongly updated are highlighted in orange.

Sample 1 is one of the failed examples induced by misclassification. This sample is non-heuristic-prone, but with a high ReplaceRate (0.75) and MatchedLevelsNum (3), meanwhile, with relatively small values for NonLetterCount (0), LongestChangedSeq (2), and TotalChangedNum (3). Thus, it is undoubtedly predicted as heuristic-prone and allocated to HebCUP Side to handle. As one of the potential reasons, the misclassification can be summarized as a lack of discriminative features, the currently proposed features cannot cover all samples' characteristics, thus fails on some of them. On the other hand, this sample also exposes one of the weaknesses of HebCUP. It has difficulties conducting *correct synchronizations* when the replacement pairs cover the tokens that do not need to be updated in old comments. For example, the replacement pair found by HebCUP in this sample is *key:* "value" and *value:* "Appearance". However, there are two tokens of "value" in the old comment, where the first one should be kept intact, and the second one should be updated. In this case, HebCUP cannot tell and uniformly replaced them with "Appearance".

Fundamentally, the second situation can be attributed to the difficulties for some of the CCI samples that the current code-comment synchronizers cannot handle. As we mentioned in Section 4.3, we partition the samples into heuristic-prone and non-heuristic-prone according to the code-comment synchronization results of HebCUP. Thus, in category prediction, samples that cannot be correctly synchronized by HebCUP are almost allocated to CUP, where though CUP has a more powerful generalization capability than HebCUP, there is still a great number of them that are hard to be correctly synchronized via CUP. As such, the failure of the second situation occurs. For example, the reference comment (i.e., new comment) in Sample 2 has a series of tokens (e.g., “together”, “with”, “the”, “trade”, and “date”) that are not shown in both old and new code. Thus, it is hard to infer the *correct synchronization* for the old comment, even for CUP with relatively powerful generalization capability. When we refer to the file associated with this sample, we find “tradedPrice” in the new code is an object of class “TradedPrice” which extends the class “Optional” and has two values, the one is “tradeDate,” and the other one is “price”. Therefore, CCI samples like this are nearly impossible for current state-of-the-art code-comment synchronizers to handle successfully due to the unseen information outside the method. Therefore, combining context information may be a potential solution for these tricky samples.

8.2 Implications

Implications for professionals: As we mentioned in Section 1, inconsistent or obsolete comments (i.e., bad comments) are enormous risks during the program development and code maintenance, which may inject unintended bugs and hinder program comprehension. On the other hand, such kinds of bad comments are always ignored by developers and manually modifying them is a labor-intensive work. Hence, accurately and efficiently synchronizing comments with code changes in an automatic manner has become vitally important. However, CCI samples in practice are complicated. For example, some change contents in comments are not shown in the corresponding code changes, such cases (e.g., Sample 1 in Figure 7) indeed are hard to design handcrafted rules to cover. Whereas other samples may have obvious replacement pairs between old and new code, but due to some interference factors (e.g., high-frequency tokens [24, 85, 96]), those deep learning-based approaches may fail to conduct the *correct synchronization* (e.g., Sample 2 in Figure 7). Instead, heuristic-based approaches can easily handle them. To this end, this article provides a potential solution, which is classifying CCI samples before synchronizing them with code changes.

In practice, professionals can leverage both the deep learning-based (i.e., CUP) and heuristic-based approach (i.e., HebCUP) in code-comment synchronization via a category prediction for each CCI sample in advance. On the other hand, the classification is based on a machine learning model (i.e., lightGBM) and our proposed five features, thus, the classification is also very efficient. According to our statistics on the 9,204 testing samples, the category prediction costs each of them only 0.063 seconds at most, showing that the category prediction phase can be totally transparent for professionals. However, as presented in this article, the performance of code-comment synchronization can be largely boosted. Therefore, we claim that it is substantially beneficial for professionals to adopt CBS for code-comment synchronization in their program development and maintenance.

Implications for researchers: In this article, CUP and HebCUP are combined via category prediction, and the extensive experiment results have proved the effectiveness of such a combination that relies on the prediction for model proneness of CCI samples. Therefore, as the first attempt, this work testifies the feasibility of a new direction that is different from improving a single general model in code-comment synchronization field. To this end, this article can be seen as a footstone and vitally important, because its methodology can be further popularized and iterated with the

development of the code-comment synchronization field. For example, with more and more code-comment synchronizers proposed in the future, the CCI samples can be further subdivided and continue to improve the performance via multi-categorical classification and combination.

On the other hand, based on our definition and partition of heuristic-prone and non-heuristic-prone samples, designing models for samples of specific categories became actionable and promising to further improve the performance. For example, CCI samples whose comment change contents are not shown in the code changes are tough to handle even for current deep learning approaches, i.e., CUP, as we mentioned in Section 8.1. Therefore, the proposal of solutions for such samples can be a way forward in the field of code-comment synchronization, such as combining the context information around the code change snippets.

Finally, as we always highlight in this article, promoting the performance of category prediction is also a helpful research direction. One of the most promising perspectives is exploring more discriminative features to differentiate between categories because such a perspective can effectively improve the classification results while ensuring the efficiency of code-comment synchronization.

9 THREATS TO VALIDITY

In this section, we clarify the threats to internal, external, and construct validity.

9.1 Internal Validity

Similar to a vast number of studies [17, 34, 41, 51, 100], the implementation for approaches and evaluation metrics can be one of the threats to the internal validity in this article. For baselines replication, we directly used the source code published in their original papers. For the implementation of the evaluation metrics that appeared in baselines, we adopted Lin et al.'s [50] code to implement Accuracy and Recall@5, since the case and punctuation marks in the comments indeed do not affect developers' comprehension to code comments. Then, we adopted Liu et al.'s [53] code to implement AED, RED, and BLEU-4 except we ignored the case, since Lin et al. [50] did not publish the source code of these metrics. Besides, to reduce the threats of the implementation of our proposals (e.g., CBS framework and ESS Ratio), we double-checked and carefully tested the code. Besides, we also published the whole project of CBS to enable other researchers to replicate and extend our work. Therefore, the threat of implementation is limited.

On the other hand, artifacts adopted in this article may be another threat to the internal validity, which is also referred to as instrumentation threats [9]. We adopted a series of third-party tools across the whole work, such as TensorFlow [8] and sci-kit learn toolkit [65] for model construction, and the *diff* tool [4] for code change alignment. Since the above artifacts are continuously tested and widely used in both academia and industries, we believe the threats of instrumentation to the internal validity can be minimized.

Besides, CCI samples with relatively large comment changes were filtered out by Liu et al. [53] because they argued that these filtered samples should be handled by code summarization models to generate new comments instead of code-comment synchronization models. However, the filtering criterion was based on their experience, leading to the adopted dataset may not cover all the code-comment synchronization situations, thus, the selection of CCI samples may affect the internal validity.

Last but not least, due to the restriction of time and computation resources, the tuning scopes of hyper-parameters for base classifiers in this article are limited. Thus, other hyper-parameter settings of lighGBM or other classifiers may yield better results, which may become one of the threats to internal validity.

9.2 External Validity

Threats to external validity focus on the experimental dataset used in this work. The dataset is only built from Java repositories and consists of comments only in the granularity of the method level, which may not be representative of all programming languages and comment types. Nevertheless, Java is undoubtedly one of the most universal and popular programming languages, while method comments are also widely studied and referred to by researchers in other fields of program comprehension, such as comment generation [33, 34, 47, 93] and comment classification [63, 64]. In addition, the dataset is extracted from the 1,496 popular Github repositories, which reflects a relatively strong representation of this dataset. On the other hand, our proposed CBS approach is also language-agnostic and can be easily applied to projects of other programming languages. Therefore, there is little threat to external validity.

9.3 Construct Validity

Threats to construct validity are related to the adoption of the evaluation metrics. In this paper, we employ Accuracy, Recall@5, AED, RED, BLEU-4, and ESS ratio to evaluate the performance of code-comment synchronization approaches. The first five metrics are summarized from the works of Liu and Lin et al. [50, 53] whereas the last one is proposed in this work as complementation. Accuracy and Recall@5 evaluate to what extent an approach can synchronize comments correctly. AED and RED measure the average edits that developers need to perform to correctly update comment after using the code-comment synchronizers. ESS ratio is a complementary metric for AED and RED, which measures the ratio of samples whose edit distances are reduced after the synchronization. BLEU-4 is an evaluation metric widely used in machine translation tasks [31, 66, 83, 99] and related software engineering studies [10, 33, 34, 74], which can provide us with an effective measurement of comment quality that is generated by code-comment synchronization approaches. Although these automated evaluation metrics cannot perfectly represent the human judgment, they can provide the quick and quantitative assessment of code-comment synchronization approaches.

10 RELATED WORK

In this section, we firstly discuss prior works related to code-comment synchronization, including code summarization, comment classification, and inconsistent comment detection. Then, we introduce previous studies on code-comment synchronization itself.

10.1 Code Summarization

Code summarization (a.k.a. comment generation) aims to generate a natural language description of source code for code comprehension [29, 55, 69]. In recent years, more and more researchers have devoted themselves to this field, and proposed many automatic code summarization approaches [12, 33, 34, 49, 54, 84, 89, 93]. Iyer et al. [36] for the first time proposed a deep learning method to summarize C# code snippets and SQL queries. Hu et al. [33, 34] and LeClair et al. [47] argued that the structure information of source code should be considered. The former proposed a **Structural-Based Traversal (SBT)** method to convert the **Abstract Syntax Tree (AST)** into a specially formatted sequence, while the latter proposed to adopt **Graph Neural Network (GNN)** to extract the local semantic information from the AST. Ahmad et al. [10] improved the performance of code summarization models by using Transformer rather than **Recurrent Neural Network (RNN)** to capture the long-range dependencies between code tokens. Zhang et al. [96] and Wei et al. [86] proposed to combine retrieval-based and deep learning-based methods to generate code comments. The former retrieved the two most similar source code to the target sample from the training set to assist the prediction of the neural model, while the latter retrieved the

comment of the most similar source code as an exemplar to guide the neural model to output a more accurate comment. Chen et al. [17] proposed to firstly predict the comment category of source code, and then choose the most suitable code summarization models for inferred categories to generate code comments. Both code summarization and code-comment synchronization focus on facilitating the program comprehension for developers, but the former generates code comments from scratch, targeting code snippets without any comments. However, the latter synchronizes pre-existing comments with corresponding code changes [53].

10.2 Comment Classification

Besides the source code, code comments have been considered as the most essential software artifacts for program comprehension and maintenance [20, 25, 26, 71]. Nevertheless, comments with different characteristics have different goals and target audiences, and analyzing comments in a single manner hinders the empirical understanding of both comments and associated code [63, 64]. Researchers have proposed to classify code comments into different categories to help developers comprehend the source code better [58, 63, 64, 67, 82, 95, 97]. Haouari et al. [30] proposed the taxonomy of comments and used it to conduct their analysis based on 39 programmer subjects and three open-source projects. Steidel et al. [75] adopted machine learning methods to classify comments into seven categories, and further analyze and evaluate the quality of code comments. Zhai et al. [95] constructed a comprehensive comment taxonomy to classify comments into appropriate perspectives and granularity levels. In summary, despite the fact that comment classification has been proposed for many years and extensively applied in various fields of software engineering, classification for CCI samples has not been investigated yet. To the best of our knowledge, we are the first work to analyse the categories of CCI samples and their application in code-comment synchronization.

10.3 Inconsistent Comment Detection

Researchers have investigated the inconsistent comment detection to improve software maintainability and reduce bugs when code-comment evolves [51, 59, 68, 76, 77, 79, 80, 98]. Seminal works in this field are conducted by Tan et al. [77–79]. They first proposed an inconsistent comment detection approach named *iComment* [77], which combined **Natural Language Processing (NLP)**, machine learning, and program analysis techniques. In the follow-up works, they investigated the detection of code-comment inconsistencies concerning the specific concepts of lock mechanism [78], function calls [78], and concurrency related interrupts [79]. Stulova et al. [76] designed a technique and a tool, *upDoc*, which built a map between the code and its comment, ensuring that changes in the code match the changes in the respective comment. Liu et al. [51] proposed a machine learning-based method to check whether the comments should be changed with the code evolution. In addition, Cimasa et al. [18] adopted word embedding techniques to detect the incoherence between source code and their associated comment. Their extensive experiments demonstrate their method is more efficient in terms of execution time while maintaining performance very close to the baseline. Moreover, Corazza et al. [19] created a publicly available Java dataset consisting of 3,636 methods in three open-source software applications to investigate coherence between the source code and associated comments. Besides, their adoption of **Support Vector Machine (SVM)** with tf-idf has also been proved to be effective in code-comment incoherence detection. The above-mentioned studies focus on code-comment inconsistency detection, while code-comment synchronization aims to automatically synchronize comments when the associated code changes, thereby avoiding the introduction of inconsistent comments.

10.4 Code-Comment Synchronization

Code-comment synchronization is an emerging research field started from the year of 2020. Until now, a total of three works [50, 53, 60] have been conducted to solve the research problem. Liu et al. [53] and Panthaplackel et al. [60] almost simultaneously for the first time proposed their own solution for code-comment synchronization with the code evolution. The former proposed an LSTM-based NMT model, which integrates the information from the old code, new code, and old comments to predict the corresponding new comments; while the latter proposed a similar GRU-based NMT model. Subsequently, Lin et al. [50] proposed HebCUP, for which they designed a series of heuristic rules to perform the token-level replacements based on old comments. Since Panthaplackel et al.'s [60] approach can only synchronize the comments of the functions with return statements, and their proposed heuristic features incorporated in their approach cannot apply to other kinds of functions, which causes that their approach is not generalized enough and applicable as the complementary of HebCUP and hard to compare with other approaches under the same criteria. Therefore, in our work, we only try to combine CUP and HebCUP to improve the code-comment synchronization performance.

11 CONCLUSION

In this paper, we investigate the performance of two state-of-the-art code-comment synchronization approaches, i.e., the deep learning-based CUP and the heuristic-based HebCUP on the unified evaluation criteria. We find that the two approaches of different mechanisms have their heterogeneous characteristics and are prone to accurately synchronize different kinds of **Code-Comment Inconsistent (CCI)** samples. Motivated by this finding, we define two categories (i.e., heuristic-prone and non-heuristic-prone) for the CCI samples and propose five features to assist category prediction. Then, we propose a composite approach named CBS, which combines the advantages of CUP and HebCUP. CBS firstly constructs a Multi-Subsets Ensemble Learning (MSEL) classification model based on the training samples. Then, CBS employs the trained MSEL model to predict whether the new CCI sample can be successfully handled by HebCUP (i.e., the predicted category of the sample is heuristic-prone). If so, CBS allocates the sample to HebCUP to conduct the code-comment synchronization; otherwise, the sample is assigned to CUP. The experimental results show that our composite approach CBS outperforms CUP and HebCUP that do not consider the category prediction for CCI samples by 8.53%–38.41% in terms of Accuracy, by 10.52%–35.16% in terms of Recall@5, by 1.24%–4.58% in terms of AED and RED, by 1.62%–1.67% in terms of BLEU-4, and by 12.09%–26.69% in terms of ESS ratio. In summary, our research emphasizes that considering automatically category classification for CCI samples before synchronizing their comments is conducive and can significantly boost the performance of code-comment synchronization.

REFERENCES

[1] 2018. A Commit in Apache Wicket. https://github.com/apache/wicket/pull/283/commits/8dcf2e34927e0c164235f5bea79c7026d22192dc. (Accessed on 02/24/2022).
[2] 2019. A Commit in Google Nomulus. https://github.com/google/nomulus/commit/cf507dad6d7bfc9e30eb520da0c08a75d054b2bd. (Accessed on 02/24/2022).
[3] 2022. apache/hive: Apache Hive. https://github.com/apache/hive. (Accessed on 02/25/2022).
[4] 2022. Difflib – Helpers for Computing Deltas – Python 3.10.2 Documentation. https://docs.python.org/3/library/difflib.html. (Accessed on 03/07/2022).
[5] 2022. Facebook/fresco: An Android Library for Managing Images and the Memory They Use. https://github.com/facebook/fresco. (Accessed on 02/25/2022).
[6] 2022. GitHub. https://github.com/. (2022). (Accessed on 02/25/2022).
[7] 2022. Google/nomulus: Top-level Domain Name Registry Service on Google App Engine. https://github.com/google/nomulus. (Accessed on 02/25/2022).

[8] 2022. TensorFlow. https://www.tensorflow.org/. (Accessed on 02/26/2022).

[9] Silvia Abrahao, Carmine Gravino, Emilio Insfran, Giuseppe Scanniello, and Genoveffa Tortora. 2012. Assessing the effectiveness of sequence diagrams in the comprehension of functional requirements: Results from a family of five experiments. *IEEE Transactions on Software Engineering* 39, 3 (2012), 327–342.

[10] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2020. A transformer-based approach for source code summarization. *arXiv preprint arXiv:2005.00653* (2020).

[11] Saad Albawi, Tareq Abed Mohammed, and Saad Al-Zawi. 2017. Understanding of a convolutional neural network. In *2017 International Conference on Engineering and Technology (ICET)*. IEEE, 1–6.

[12] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2018. code2seq: Generating sequences from structured representations of code. *arXiv preprint arXiv:1808.01400* (2018).

[13] William H. Beyer. 2019. *Handbook of Tables for Probability and Statistics*. CRC Press.

[14] Xavier Bouthillier and Gaël Varoquaux. 2020. *Survey of Machine-learning Experimental Methods at NeurIPS2019 and ICLR2020*. Ph.D. Dissertation. Inria Saclay Ile de France.

[15] Leo Breiman. 2001. Random forests. *Machine Learning* 45, 1 (2001), 5–32.

[16] Leo Breiman, Jerome Friedman, Charles J. Stone, and Richard A. Olshen. 1984. *Classification and Regression Trees*. CRC Press.

[17] Qiuyuan Chen, Xin Xia, Han Hu, David Lo, and Shanping Li. 2021. Why my code summarization model does not work: Code comment improvement with category prediction. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 30, 2 (2021), 1–29.

[18] Alfonso Cimasa, Anna Corazza, Carmen Coviello, and Giuseppe Scanniello. 2019. Word embeddings for comment coherence. In *2019 45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE, 244–251.

[19] Anna Corazza, Valerio Maggio, and Giuseppe Scanniello. 2018. Coherence of comments and method implementations: A dataset and an empirical investigation. *Software Quality Journal* 26, 2 (2018), 751–777.

[20] Sergio Cozzetti B. de Souza, Nicolas Anquetil, and Káthia M. de Oliveira. 2005. A study of the documentation essential to software maintenance. In *Proceedings of the 23rd Annual International Conference on Design of Communication: Documenting & Designing for Pervasive Information*. 68–75.

[21] Thomas J. DiCiccio and Bradley Efron. 1996. Bootstrap confidence intervals. *Statistical Science* 11, 3 (1996), 189–228.

[22] J. A. Ferreira, A. H. Zwinderman, et al. 2006. On the Benjamini–Hochberg method. *Annals of Statistics* 34, 4 (2006), 1827–1849.

[23] Markus Freitag and Yaser Al-Onaizan. 2017. Beam search strategies for neural machine translation. *arXiv preprint arXiv:1702.01806* (2017).

[24] Cuiyun Gao, Wenjie Zhou, Xin Xia, David Lo, Qi Xie, and Michael R. Lyu. 2020. Automating app review response generation based on contextual knowledge. *CoRR* abs/2010.06301 (2020). arXiv:2010.06301 https://arxiv.org/abs/2010.06301.

[25] Verena Geist, Michael Moser, Josef Pichler, Stefanie Beyer, and Martin Pinzger. 2020. Leveraging machine learning for software redocumentation. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 622–626. https://doi.org/10.1109/SANER48275.2020.9054838

[26] Mingyang Geng, Shangwen Wang, Dezun Dong, Shanzhi Gu, Fang Peng, Weijian Ruan, and Xiangke Liao. 2022. Fine-grained code-comment semantic interaction analysis. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension (ICPC)*.

[27] Sergio González, Salvador García, Javier Del Ser, Lior Rokach, and Francisco Herrera. 2020. A practical tutorial on bagging and boosting based ensembles for machine learning: Algorithms, software tools, performance study, practical perspectives and opportunities. *Information Fusion* 64 (2020), 205–237. https://doi.org/10.1016/j.inffus.2020.07.007

[28] Kevin Gurney. 1997. *An Introduction to Neural Networks*. CRC Press.

[29] Sonia Haiduc, Jairo Aponte, and Andrian Marcus. 2010. Supporting program comprehension with source code summarization. In *2010 ACM/IEEE 32nd International Conference on Software Engineering*, Vol. 2. IEEE, 223–226.

[30] Dorsaf Haouari, Houari Sahraoui, and Philippe Langlais. 2011. How good is your comment? A study of comments in Java programs. In *2011 International Symposium on Empirical Software Engineering and Measurement*. 137–146. https://doi.org/10.1109/ESEM.2011.22

[31] Di He, Yingce Xia, Tao Qin, Liwei Wang, Nenghai Yu, Tie-Yan Liu, and Wei-Ying Ma. 2016. Dual learning for machine translation. *Advances in Neural Information Processing Systems* 29 (2016), 820–828.

[32] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural Computation* 9, 8 (1997), 1735–1780.

[33] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018. Deep code comment generation. In *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*. IEEE, 200–210.

[34] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2020. Deep code comment generation with hybrid lexical and syntactical information. *Empirical Software Engineering* 25, 3 (2020), 2179–2217.
[35] Walid M. Ibrahim, Nicolas Bettenburg, Bram Adams, and Ahmed E. Hassan. 2012. On the relationship between comment update practices and software bugs. *Journal of Systems and Software* 85, 10 (2012), 2293–2304.
[36] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. Summarizing source code using a neural attention model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 2073–2083.
[37] Liangxiao Jiang, Dianhong Wang, Zhihua Cai, and Xuesong Yan. 2007. Survey of improving Naive Bayes for classification. In *International Conference on Advanced Data Mining and Applications*. Springer, 134–145.
[38] Mira Kajko-Mattsson. 2005. A survey of documentation practice within corrective maintenance. *Empirical Software Engineering* 10, 1 (2005), 31–55.
[39] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. 2017. LightGBM: A highly efficient gradient boosting decision tree. *Advances in Neural Information Processing Systems* 30 (2017), 3146–3154.
[40] Jessica Keyes. 2002. *Software Engineering Handbook*. Auerbach Publications.
[41] Dong Jae Kim, Nikolaos Tsantalis, Tse-Hsun Chen, and Jinqiu Yang. 2021. Studying test annotation maintenance in the wild. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 62–73. https://doi.org/10.1109/ICSE43902.2021.00019
[42] Serkan Kiranyaz, Onur Avci, Osama Abdeljaber, Turker Ince, Moncef Gabbouj, and Daniel J. Inman. 2021. 1D convolutional neural networks and applications: A survey. *Mechanical Systems and Signal Processing* 151 (2021), 107398.
[43] Carsten Kolassa, Dirk Riehle, and Michel A. Salim. 2013. The empirical commit frequency distribution of open source projects. In *Proceedings of the 9th International Symposium on Open Collaboration*. 1–8.
[44] F. D. C. Kraaikamp and H. L. L. Meester. 2005. A Modern Introduction to Probability and Statistics. (2005).
[45] Adrian Kuhn, Stéphane Ducasse, and Tudor Gîrba. 2007. Semantic clustering: Identifying topics in source code. *Information and Software Technology* 49, 3 (2007), 230–243.
[46] Max Kuhn. 2008. Building predictive models in R using the caret package. *Journal of Statistical Software* 28 (2008), 1–26.
[47] Alexander LeClair, Sakib Haque, Lingfei Wu, and Collin McMillan. 2020. Improved code summarization via a graph neural network. In *Proceedings of the 28th International Conference on Program Comprehension*. 184–195.
[48] Joseph Lev et al. 1949. The point biserial coefficient of correlation. *Annals of Mathematical Statistics* 20, 1 (1949), 125–126.
[49] Yuding Liang and Kenny Zhu. 2018. Automatic generation of text descriptive comments for code blocks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 32.
[50] Bo Lin, Shangwen Wang, Kui Liu, Xiaoguang Mao, and Tegawendé F. Bissyandé. 2021. Automated comment update: How far are we?. In *2021 29th IEEE/ACM International Conference on Program Comprehension (ICPC)*. IEEE, 36–46.
[51] Zhiyong Liu, Huanchao Chen, Xiangping Chen, Xiaonan Luo, and Fan Zhou. 2018. Automatic detection of outdated comments during code changes. In *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, Vol. 01. 154–163. https://doi.org/10.1109/COMPSAC.2018.00028
[52] Zhongxin Liu, Xin Xia, David Lo, Meng Yan, and Shanping Li. 2021. Just-in-time obsolete comment detection and update. *IEEE Transactions on Software Engineering* (2021), 1–1. https://doi.org/10.1109/TSE.2021.3138909
[53] Zhongxin Liu, Xin Xia, Meng Yan, and Shanping Li. 2020. Automating just-in-time comment updating. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 585–597.
[54] Yangyang Lu, Zelong Zhao, Ge Li, and Zhi Jin. 2017. Learning to generate comments for API-based code snippets. In *Software Engineering and Methodology for Emerging Domains*. Springer, 3–14.
[55] Paul W. McBurney and Collin McMillan. 2015. Automatic source code summarization of context for Java methods. *IEEE Transactions on Software Engineering* 42, 2 (2015), 103–119.
[56] Patrick E. McKnight and Julius Najab. 2010. Mann-Whitney U test. *The Corsini Encyclopedia of Psychology* (2010), 1–1.
[57] Gonzalo Navarro. 2001. A guided tour to approximate string matching. *ACM Computing Surveys (CSUR)* 33, 1 (2001), 31–88.
[58] Yoann Padioleau, Lin Tan, and Yuanyuan Zhou. 2009. Listening to programmers− taxonomies and characteristics of comments in operating system code. In *2009 IEEE 31st International Conference on Software Engineering*. 331–341. https://doi.org/10.1109/ICSE.2009.5070533
[59] Sheena Panthaplackel, Junyi Jessy Li, Milos Gligoric, and Raymond J. Mooney. 2020. Deep just-in-time inconsistency detection between comments and source code. *arXiv preprint arXiv:2010.01625* (2020).
[60] Sheena Panthaplackel, Pengyu Nie, Milos Gligoric, Junyi Jessy Li, and Raymond Mooney. 2020. Learning to update natural language comments based on code changes. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. 1853–1868.

[61] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. BLEU: A method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*. 311–318.

[62] David Lorge Parnas. 2011. Precise documentation: The key to better software. In *The Future of Software Engineering*. Springer, 125–148.

[63] Luca Pascarella and Alberto Bacchelli. 2017. Classifying code comments in Java open-source software systems. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 227–237.

[64] Luca Pascarella, Magiel Bruntink, and Alberto Bacchelli. 2019. Classifying code comments in Java software systems. *Empirical Software Engineering* 24, 3 (2019), 1499–1537.

[65] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. 2011. Scikit-learn: Machine learning in Python. *The Journal of Machine Learning Research* 12 (2011), 2825–2830.

[66] Martin Popel, Marketa Tomkova, Jakub Tomek, Łukasz Kaiser, Jakob Uszkoreit, Ondřej Bojar, and Zdeněk Žabokrtskỳ. 2020. Transforming machine translation: A deep learning system reaches news translation quality comparable to human professionals. *Nature Communications* 11, 1 (2020), 1–15.

[67] Pooja Rani, Sebastiano Panichella, Manuel Leuenberger, Andrea Di Sorbo, and Oscar Nierstrasz. 2021. How to identify class comment types? A multi-language approach for class comment classification. *Journal of Systems and Software* 181 (2021), 111047.

[68] Inderjot Kaur Ratol and Martin P. Robillard. 2017. Detecting fragile comments. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 112–122.

[69] Paige Rodeghero, Cheng Liu, Paul W. McBurney, and Collin McMillan. 2015. An eye-tracking study of Java programmers and application to source code summarization. *IEEE Transactions on Software Engineering* 41, 11 (2015), 1038–1054.

[70] Hinrich Schütze, Christopher D. Manning, and Prabhakar Raghavan. 2008. *Introduction to Information Retrieval*. Vol. 39. Cambridge University Press Cambridge.

[71] Yusuke Shinyama, Yoshitaka Arahori, and Katsuhiko Gondow. 2018. Analyzing code comments to boost program comprehension. In *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*. 325–334. https://doi.org/10.1109/APSEC.2018.00047

[72] Kamilya Smagulova and Alex Pappachen James. 2019. A survey on LSTM memristive neural network architectures and applications. *The European Physical Journal Special Topics* 228, 10 (2019), 2313–2324.

[73] Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori Pollock, and K. Vijay-Shanker. 2010. Towards automatically generating summary comments for Java methods. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*. 43–52.

[74] Sean Stapleton, Yashmeet Gambhir, Alexander LeClair, Zachary Eberhart, Westley Weimer, Kevin Leach, and Yu Huang. 2020. A human study of comprehension and code summarization. In *Proceedings of the 28th International Conference on Program Comprehension*. 2–13.

[75] Daniela Steidl, Benjamin Hummel, and Elmar Juergens. 2013. Quality analysis of source code comments. In *2013 21st International Conference on Program Comprehension (ICPC)*. 83–92. https://doi.org/10.1109/ICPC.2013.6613836

[76] Nataliia Stulova, Arianna Blasi, Alessandra Gorla, and Oscar Nierstrasz. 2020. Towards detecting inconsistent comments in Java source code automatically. In *2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 65–69.

[77] Lin Tan, Ding Yuan, Gopal Krishna, and Yuanyuan Zhou. 2007. /* iComment: Bugs or bad comments?*. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*. 145–158.

[78] Lin Tan, Ding Yuan, and Yuanyuan Zhou. 2007. Hotcomments: How to make program comments more useful?. In *HotOS*.

[79] Lin Tan, Yuanyuan Zhou, and Yoann Padioleau. 2011. aComment: Mining annotations from comments and code to detect interrupt related concurrency bugs. In *2011 33rd International Conference on Software Engineering (ICSE)*. IEEE, 11–20.

[80] Shin Hwei Tan, Darko Marinov, Lin Tan, and Gary T. Leavens. 2012. @tComment: Testing Javadoc comments to detect comment-code inconsistencies. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. 260–269. https://doi.org/10.1109/ICST.2012.106

[81] Chakkrit Tantithamthavorn, Ahmed E. Hassan, and Kenichi Matsumoto. 2020. The impact of class rebalancing techniques on the performance and interpretation of defect prediction models. *IEEE Transactions on Software Engineering* 46, 11 (2020), 1200–1219.

[82] Betty Van Aken, Julian Risch, Ralf Krestel, and Alexander Löser. 2018. Challenges for toxic comment classification: An in-depth error analysis. *arXiv preprint arXiv:1809.07572* (2018).

[83] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.), Vol. 30. Curran Associates, Inc. https://proceedings.neurips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf.

[84] Yao Wan, Zhou Zhao, Min Yang, Guandong Xu, Haochao Ying, Jian Wu, and Philip S. Yu. 2018. Improving automatic source code summarization via deep reinforcement learning. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 397–407.

[85] Haoye Wang, Xin Xia, David Lo, Qiang He, Xinyu Wang, and John Grundy. 2021. Context-aware retrieval-based deep commit message generation. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 30, 4 (2021), 1–30.

[86] Bolin Wei, Yongmin Li, Ge Li, Xin Xia, and Zhi Jin. 2020. Retrieve and refine: Exemplar-based neural comment generation. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 349–360.

[87] Fengcai Wen, Csaba Nagy, Gabriele Bavota, and Michele Lanza. 2019. A large-scale empirical study on code-comment inconsistencies. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE, 53–64.

[88] Frank Wilcoxon. 1992. Individual comparisons by ranking methods. In *Breakthroughs in Statistics*. Springer, 196–202.

[89] Edmund Wong, Taiyue Liu, and Lin Tan. 2015. CloCom: Mining existing source code for automatic comment generation. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 380–389.

[90] Fei Wu, Xiao-Yuan Jing, Shiguang Shan, Wangmeng Zuo, and Jing-Yu Yang. 2017. Multiset feature learning for highly imbalanced data classification. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 31.

[91] Xin Xia, Lingfeng Bao, David Lo, Zhenchang Xing, Ahmed E. Hassan, and Shanping Li. 2017. Measuring program comprehension: A large-scale field study with professionals. *IEEE Transactions on Software Engineering* 44, 10 (2017), 951–976.

[92] Zhen Yang. 2022. yz1019117968/TOSEM-22-CBS: Source Code for "On the Significance of Category Prediction for Code-Comment Synchronization". https://github.com/yz1019117968/TOSEM-22-CBS. (Accessed on 05/04/2022).

[93] Zhen Yang, Jacky Keung, Xiao Yu, Xiaodong Gu, Zhengyuan Wei, Xiaoxue Ma, and Miao Zhang. 2021. A multi-modal transformer-based code summarization approach for smart contracts. In *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*. 1–12. https://doi.org/10.1109/ICPC52881.2021.00010

[94] Tong Yu and Hong Zhu. 2020. Hyper-parameter optimization: A review of algorithms and applications. *arXiv preprint arXiv:2003.05689* (2020).

[95] Juan Zhai, Xiangzhe Xu, Yu Shi, Guanhong Tao, Minxue Pan, Shiqing Ma, Lei Xu, Weifeng Zhang, Lin Tan, and Xiangyu Zhang. 2020. CPC: Automatically classifying and propagating natural language comments via program analysis. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 1359–1371.

[96] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, and Xudong Liu. 2020. Retrieval-based neural source code summarization. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 1385–1397.

[97] Junli Zhang, Xuewei Zhao, Ming Xian, Chuan Dong, and Shaomin Shuang. 2018. Folic acid-conjugated green luminescent carbon dots as a nanoprobe for identifying folate receptor-positive cancer cells. *Talanta* 183 (2018), 39–47. https://doi.org/10.1016/j.talanta.2018.02.009

[98] Yu Zhou, Ruihang Gu, Taolue Chen, Zhiqiu Huang, Sebastiano Panichella, and Harald Gall. 2017. Analyzing APIs documentation and code to detect directive defects. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 27–37.

[99] Jinhua Zhu, Yingce Xia, Lijun Wu, Di He, Tao Qin, Wengang Zhou, Houqiang Li, and Tie-Yan Liu. 2020. Incorporating BERT into neural machine translation. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net. https://openreview.net/forum?id=Hyl7ygStwB

[100] Qihao Zhu, Zeyu Sun, Yuan-an Xiao, Wenjie Zhang, Kang Yuan, Yingfei Xiong, and Lu Zhang. 2021. A syntax-guided edit decoder for neural program repair. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 341–353.

Received 7 December 2021; revised 24 March 2022; accepted 27 April 2022