Just-In-Time Obsolete Comment Detection and Update

Zhongxin Liu, Xin Xia, David Lo, Meng Yan, Shanping Li

Abstract—Comments are valuable resources for the development, comprehension and maintenance of software. However, while changing code, developers sometimes neglect the evolution of the corresponding comments, resulting in obsolete comments. Such obsolete comments can mislead developers and introduce bugs in the future, and are therefore detrimental. We notice that by detecting and updating obsolete comments in time with code changes, obsolete comments can be effectively reduced and even avoided. We refer to this task as Just-In-Time (JIT) Obsolete Comment Detection and Update. In this work, we propose a two-stage framework named **CUP^2** (**Two**-stage **C**omment **UP**dater) to automate this task. CUP^2 consists two components, i.e., an **O**bsolete **C**omment **D**etector named **OCD** and a **C**omment **UP**dater named **CUP**, each of which relies on a distinct neural network model to perform detection (updates). Specifically, given a code change and a corresponding comment, CUP^2 first leverages OCD to predict whether this comment should be updated. If the answer is yes, CUP will be used to generate the new version of the comment automatically. To evaluate CUP^2, we build a large-scale dataset with over 4 million code-comment change samples. Our dataset focuses on method-level code changes and updates on method header comments considering the importance and widespread use of such comments. Evaluation results show that 1) both OCD and CUP outperform their baselines by significant margins, and 2) CUP^2 performs better than a rule-based baseline. Specifically, the comments generated by CUP^2 are identical to the ground truth for 41.8% of the samples that are predicted to be positive by OCD. We believe CUP^2 can help developers detect obsolete comments, better understand where and how to update obsolete comments and reduce their edits on obsolete comment updates.

Index Terms—code-comment co-evolution, obsolete comment detection, comment update

1 Introduction

Code comments are invaluable assets to software projects. Various information of code, such as why and how a function is implemented, how to use an API, what is the relation between two code snippets, and how a code segment evolves [1], [2], [3] is recorded in code comments. Such information can help understand source code and facilitate the communication between developers [4], [5], and can therefore help software development and maintenance. As shown by a prior study [6], besides source code, developers regard comments as the most essential artifacts for understanding and maintaining a software system [6].

As important references for source code, comments need to co-evolve with the corresponding code snippets. However, in practice, developers may neglect updating comments while changing source code, resulting in obsolete comments. For example, Table 1 presents an obsolete comment collected from Apache Kafka [7]. We can see that the method in Table 1 registers the metrics of the producer, consumer, and admin clients. However, only the consumer clients are recorded in the corresponding comment, which means this comment needs to be updated and is an obsolete comment. This obsolete comment had existed in the code base for eight months before being fixed. During this period, such comments can mislead developers, complicate code reviews, lead to the introduction of bugs, and have negative effects on the robustness of the software system [8], [9], [10], [11], [12]. Therefore, it is necessary to fix obsolete comments in time or even avoid their introduction.

To gain more insights about obsolete comments, we further check the change history of the method in Table 1. The change history "tells" us that when the comment was first introduced, the code only registered the consumer

TABLE 1
A obsolete comment example

```
public Map<MetricName, ? extends Metric> metrics() {
    ...
    for (final StreamThread thread : threads) {
        result.putAll(thread.producerMetrics());
        result.putAll(thread.consumerMetrics());
        result.putAll(thread.adminClientMetrics());
    }
    ...}
```

Method Comment: Get read-only handle on global metrics registry, including streams client's own metrics plus its embedded consumer clients' metrics.

Updated Comment: Get read-only handle on global metrics registry, including streams client's own metrics plus its embedded producer, consumer and admin clients' metrics.

- *Zhongxin Liu is with the College of Computer Science and Technology and Ningbo Research Institute, Zhejiang University, China, and PengCheng Laboratory, China.*
  E-mail: liu_zx@zju.edu.cn
- *Xin Xia is with the Software Engineering Application Technology Lab, Huawei, China.*
  E-mail: xin.xia@acm.org
- *David Lo is with the School of Information Systems, Singapore Management University, Singapore*
  E-mail: davidlo@smu.edu.sg
- *Meng Yan is with the School of Big Data and Software Engineering, Chongqing University, China.*
  E-mail: mengy@cqu.edu.cn
- *Shanping Li is with the College of Computer Science and Technology, Zhejiang Univeristy, China.*
  E-mail: shan@zju.edu.cn
- *Xin Xia is the corresponding author.*

clients' metrics and the code and comment were consistent. However, two following changes added more metrics without updating the comment and made this comment an obsolete one. This finding inspires us that when developers make code changes, if a tool can automatically detect the associated comments requiring updates and update such comments correspondingly, it is possible to reduce and even avoid the introduction of obsolete comments. We refer to this task as Just-In-Time (JIT) obsolete comment detection and update.

Researchers have investigated the detection of obsolete comments. But most of them [8], [9], [11], [13], [14] only focus on the comments related to specific topics, such as lock mechanism and function calls [9] or a special comment type, e.g., TODO comments [14], instead of general comments. Others rely on manually crafted rules or features to perform obsolete comment detection [15], [16]. On the other hand, no approach has been proposed to automatically update comments with code changes. Moreover, there is a lack of effort to combine the detection and update of obsolete comments as a whole and perform automatic end-to-end obsolete comment repair.

To fill these gaps and help developers handle obsolete comments, in this work, we propose a two-stage framework named **CUP^2** (**Two**-stage **C**omment **UP**dater) to automate the JIT obsolete comment detection and update. CUP^2 consists of two components: an **O**bsolete **C**omment **D**etector named **OCD** and a **C**omment **UP**dater named **CUP**. When a developer changes a code snippet, OCD can predict whether an associated comment should be updated. For detected obsolete comments, CUP will be further used to automatically generate their updated versions.

To automate obsolete comment detection and update, an intuitive way is manually summarizing and implementing detection features and update rules of obsolete comments. However, comments are free-form texts written in natural languages, which are inherently ambiguous and unstructured. Thus, it is difficult and time-consuming to craft such features and update rules. The idea behind CUP^2 is to leverage neural network models to automatically learn the patterns of obsolete comments and comment updates from large-scale code changes, which can be extracted from massive code repositories. Specifically, we first propose a novel neural encoder to represent a code change and an associated comment as feature vectors, simultaneously. Then, OCD is composed of the neural encoder and an attention-based output layer to predict the probability that the associated comment should be updated. CUP also leverages the encoder for input representation but connects it with an RNN-based decoder to generate updated comments.

Compared to existing obsolete comment detectors [8], [9], [11], [13], [14], [15], [16], OCD targets at general comments and do not require manual efforts to craft rules or features. CUP is a neural sequence-to-sequence (seq2seq) model. Different from existing neural seq2seq models designed for other software engineering (SE) tasks [17], [18], [19], CUP takes as input both code changes and their associated comments, learns the representations of code changes and comments simultaneously, and can effectively capture the relationships between code changes and comments with the help of a unified vocabulary, the pre-trained fastText embeddings and a novel co-attention mechanism in the encoder. Also, two pointer generators are used in the decoder of CUP to deal with out-of-vocabulary (OOV) words and ease the generation of updated comments. CUP^2 combines OCD and CUP to automate JIT obsolete comment detection and update.

To evaluate the effectiveness of CUP^2 and its two components, we build a large dataset with over 4 million code-comment change samples from 1,496 popular engineered Java projects. For now, our dataset focuses on method-level code changes and updates on method header comments, because Java methods can be precisely associated with their header comments and such comments are an important type of documents that are often referred to by developers. Evaluation results show that: 1) OCD significantly outperforms its two baselines by over 17.1% in terms of Precision, Recall and F1-Score. 2) CUP performs better than its three baselines in terms of Accuracy, Recall@5, GLEU [20] and two metrics proposed by us named Average Edit Distance (AED) and Relative Edit Distance (RED) by significant margins. It replicates perfectly developer-performed comment updates in 10 times more cases than the best-performing baseline. 3) CUP^2 also outperforms a rule-based baseline by significant margins in terms of both detection and update metrics. The comments generated by CUP^2 are identical to the ground truth for 41.8% of the samples that are predicted to be positive by OCD. In addition, CUP^2 achieves a RED of 0.843, which means it can help developers better understand where and how to perform obsolete comment updates and reduce developers' edits on updating obsolete comments.

In summary, this paper makes the following contributions:

- We build a dataset with over 4 million code-comment change samples for the JIT obsolete comment detection and update task. To the best of our knowledge, so far, this is the largest dataset for this task.
- We propose a novel two-stage approach named CUP^2, which consists of a neural detector OCD and a neural updater CUP, to automate the JIT detection and update of obsolete comments. OCD and CUP leverage novel neural network models to automatically learn the patterns of obsolete comments and comment updates from large-scale code-comment change samples. They use the same encoder architecture which can represent code changes and comments simultaneously and capture the relationships between them with the help of a unified vocabulary, the pre-trained fastText embeddings and a novel co-attention mechanism. CUP also adopts pointer generators to deal with out-of-vocabulary (OOV) words and ease the generation of updated comments. As the first attempt for JIT obsolete comment detection and update, CUP^2 lays a good foundation and can inspire other researchers to tackle this important and interesting task.
- We extensively evaluate CUP^2 as well as OCD and CUP, on the constructed dataset. Evaluation results show that OCD and CUP are effective and significantly outperform their own baselines. CUP^2 can correctly update comments for 41.8% of the samples

that are predicted to be positive by OCD, help developers better understand where and how to update obsolete comments and reduce their edits on obsolete comment updates.
- We share our replication package, which includes dataset, source code and trained models, publicly at https://github.com/Tbabm/CUP2 to facilitate replication and support follow-up works.

As an extended version of our previous work [21], which proposed and evaluated CUP, this paper significantly extended our previous work in the following ways:

- We extend the evaluation in our original study with a new evaluation metric GLEU. GLEU is widely used to evaluate Grammatical Error Correction (GEC) systems in the natural language processing (NLP) field [22], [23], [24]. It is more flexible than Accuracy and Recall@5 and is shown to highly correlate with human judgments on GEC tasks [25]. We borrow this metric from the NLP field to measure the overall quality of the updated comments and complement Accuracy and Recall@5.
- We propose a neural detector named OCD to perform JIT obsolete comment prediction.
- We integrate OCD with CUP and build a two-stage approach named CUP^2 to automate the JIT detection and update of obsolete comments.
- The dataset built in our previous work only contains positive code-comment change samples, i.e., the samples where the comments get updated. In this work, we further collect a large number of negative samples from the same repositories used in our previous work, integrate them with the samples in our previous dataset, and build a large-scale dataset with over 4 million code-comment change samples for the JIT obsolete comment detection and update task.
- We evaluate the performance of the proposed detector and the two-stage approach on the built dataset. The evaluation results demonstrate their effectiveness.

The remainder of this paper is organized as follows: Section 2 formulates the problem and describes the usage scenarios of our approach. After a review of related work in Section 3, we present the details of our approach, including its overall framework, how we flatten code-comment change samples as sequences and the neural network models used in OCD and CUP in Section 4. Section 5 illustrates how we build datasets for training and evaluating CUP^2 and its two components. We describe the setup of our evaluation in Section 6 and present our evaluation results in Section 7. In Section 8, we discuss the errors that CUP^2 may encounter, our exploration to alleviate the class imbalance problem of the dataset, our exploration on the synergy between OCD and CUP, and the threats to the validity of this work. We conclude this work and point out some potential future directions in Section 9.

2 Problem and Usage Scenario

2.1 Problem Formulation

This work aims to predict and update obsolete comments given a code change. Formally, given the pre-change and post-change versions of a code snippet, namely $\boldsymbol{t}$ and $\boldsymbol{t'}$, and the pre-change and desired post-change versions of one of its associated comments, namely $\boldsymbol{x}$ and $\boldsymbol{y}$, our goal is to find a method $detect$ and a method $update$, so that:

$$detect(\boldsymbol{t}, \boldsymbol{t'}, \boldsymbol{x}) = \begin{cases} 1, & \text{if } \boldsymbol{x} \neq \boldsymbol{y} \\ 0, & \text{otherwise} \end{cases} \quad (1)$$

and if $\boldsymbol{x} \neq \boldsymbol{y}$:

$$update(\boldsymbol{t}, \boldsymbol{t'}, \boldsymbol{x}) = \boldsymbol{y}$$

For convenience, we refer to $\boldsymbol{t}$, $\boldsymbol{t'}$, $\boldsymbol{x}$ and $\boldsymbol{y}$ as *old code*, *new code*, *old comment* and *new comment*. $\boldsymbol{y}$ is the desired post-change comment, and is not available when adopting $detect$ and $update$ in practice. Since we target at updating existing old comments instead of generating new comments from scratch, keeping the format of old comments and the corresponding new comments consistent is regarded as an essential requirement for $update$.

CUP^2 leverages two neural network models, i.e., OCD and CUP, to approximate $detect$ and $update$, respectively. It requires a training set with true $\boldsymbol{y}$s for training. Based on the true $\boldsymbol{y}$s, each sample in the training set can be labeled as positive (i.e., $\boldsymbol{x} \neq \boldsymbol{y}$) or negative (i.e., $\boldsymbol{x} = \boldsymbol{y}$). The $detect$ model (i.e., OCD) will be trained on the whole training set, i.e., using both positive and negative samples, and the $update$ model (i.e., CUP) is going to be trained on the dataset with only positive samples.

2.2 Usage Scenario

The usage scenarios of CUP^2 are as follows:

First, CUP^2 can assist developers in finding, understanding and fixing obsolete method comments given code changes. In detail, when a developer makes a code change, CUP^2 can first predict whether the method comments associated with the changed methods should be updated and warn the developer of the introduced obsolete comments. For each detected obsolete comment, CUP^2 can future provide an update suggestion for it. If such suggestion is correct or partially correct, it can help developers quickly understand where and how to update comments and reduce their edits required for JIT comment update. Therefore, CUP^2 can improve the productivity of developers in avoiding obsolete comments.

CUP^2 can also help developers locate, understand and fix existing obsolete comments. For example, given a software repository, developers can first use CUP^2 to detect the comments requiring updates in each historical change. For those detected obsolete comments, CUP^2 can further generate their new versions automatically. In this way, developers can effectively identify, understand and fix existing obsolete comments instead of manually finding and modifying them.

3 Related Work

This section discusses related work concerning code-comment co-evolution, obsolete comment detection and update, and comment generation.

3.1 Code-Comment Co-Evolution

Prior studies have investigated the co-evolution between source code and code comments from different perspectives [12], [26], [27], [28], [29], [30]. For example, Jiang and Hassan [26] found the percentage of commented functions in PostgreSQL remains stable over time. Fluri et al. [27], [28] studied how source code and comments co-evolved and found that over 97% of the comment changes triggered by code changes were done in the same revisions as the associated code changes. They also highlighted that API comments are often adapted retroactively. In addition, Ibrahim et al. [12] investigated the relationship between comment update practice and software bugs in three open-source systems and found abnormal comment update behavior is a good indicator for predicting future bugs. Linares-Vásquez et al. [29] studied how developers document database usages in method comments and pointed out that the comments of database-related methods are less frequently updated than source code. Recently, Wen et al. [30] conducted a large-scale empirical study, which analyzed the chances that different code change types trigger comment updates and defined a taxonomy of the code-comment inconsistencies fixed by developers.

Different from these studies, our work aims to automatically detect and update the comments requiring updates with code changes. The empirical findings presented by previous work motivate our work to facilitate the co-evolution of code and comments and shed light into the JIT detection and update of obsolete comments.

3.2 Obsolete Comment Detection and Update

Researchers have studied the detection of obsolete comments. Most of them focused on the comments related to specific code properties or of specific types [8], [9], [11], [13], [14], [31]. Tan et al. [8], [9], [13] proposed a series of approaches to detect code-comment inconsistencies related to specific programming concepts, such as lock mechanisms [8], [9], function calls [9] and interrupts [13], using manually defined rules, NLP techniques and static program analysis. @tCOMMENT, an approach devised by Tan et al. [11], leverages heuristics and automatic test generation to check inconsistencies between Java methods and their Javadocs in terms of method parameters' tolerance to null values. Sridhara et al. [14] proposed a technique to identify obsolete TODO comments based on information retrieval, linguistics and semantics. Several studies targeted at general comments and took code changes into consideration [15], [16], [32]. For instance, Ratol and Robillard [15] proposed a rule-based approach named Fraco to detect fragile comments with respect to identifier renaming. Malik et al. [32] empirically investigated the rationale of updating the comment of a modified function from three dimensions, i.e., characteristics of the changed function, the change itself, and the time as well as code ownership. Liu et al. [16] leveraged the Random Forest algorithm and 64 manually-crafted features derived from code, comments and code-comment relationships to check whether to update a block-/line comment when its associated code snippet is changed.

The differences between these techniques and the detection stage of our approach, i.e., OCD, are three folds. First, OCD aims to detect obsolete comments with code changes, and takes as input a code change instead of only one version of code. Second, OCD is not limited to specific comment types and can handle diverse code changes. Last but not least, OCD does not require manual efforts to summarize and implement rules or features. It automatically learns how to represent code changes and comments simultaneously from massive code-comment change samples.

There are few works focusing on automatic obsolete comment update. In our previous work [21], we proposed CUP, which is used as the update stage of CUP^2, to automate JIT comment update. In a parallel work, Panthaplackel et al. [33] proposed an edit-based seq2seq model to learn to update the "@return" tags in method comments with code changes. Their model represents each code change as an XML-like edit sequence and generates similar XML-like edit sequences for comment updates. They also adopted several manually-derived features in the model, some of which are specific to the "@return" tags. Different from their work, first, CUP targets at the description sections in method comments, which are orthogonal to the "@return" tags. Second, CUP represents code changes as token-level edit sequences, generates new comments directly, and does not rely on manually-crafted features. In addition, the dataset used to evaluate CUP (with over 100K positive samples) is much larger than the one used in Panthaplackel et al.' work (with less than 10K samples).

This work combines the detection and update of obsolete comments together to provide an end-to-end solution for avoiding obsolete comments, which, to the best of our knowledge, is the first attempt in this direction.

3.3 Comment Generation

Automatic comment generation techniques may also help developers update comments by directly generating new comments from changed methods. Many previous works proposed to generate code comments using rule-based and IR-based methods [34], [35], [36], [37], [38]. For example, Sridhara et al. [34] proposed an approach to generate comments for Java methods using summary information extracted from source code and manually-defined templates. To generate a comment for a code snippet, ColCom [37] first finds similar code snippets from open source projects and then reuses and tailors their comments as output. Recently, more and more researchers leveraged probabilistic models to perform comment generation [17], [39], [40], [41], [42], [43]. For example, Iyer et al. [39] proposed a neural attention model named CODE-NN to generate summaries for C# and SQL code snippets. DeepCom, an approach proposed by Hu et al. [17], uses a structure-based traversal (SBT) method to flatten ASTs and combines such flattened sequences with an encoder-decoder model to generate comments for Java methods. In their follow-up work, Hu et al. [42] devised Hybrid-DeepCom to enhance DeepCom by combining source code and the SBT sequences together to generate comments. In a parallel work, LeClair et al. [41] proposed a similar model, which also encodes code texts and the SBT sequences using two distinct encoders, for comment generation. Chen et al. [44] proposed an ensemble approach that integrates different comment generation techniques ac-

cording to the information intention of the to-be-generated comment.

We believe the JIT comment update problem and the comment generation problem are different, because the former focuses on updating pre-existing comments instead of generating comments from scratch, and it considers both the old comment and the corresponding code change instead of only the new code. Also, the goal of this work is to combine obsolete comment detection and update, instead of generating comments.

4 Approach

The overall framework of our approach, namely CUP^2, is illustrated in Figure 1. It consists of three phases, i.e., data flattening, model training, and comment update. Specifically, we first flatten the code-comment change samples extracted from source code repositories as sequences. Each sample contains an old code, the corresponding new code, an associated old comment and the corresponding new comment. Based on whether the old and the new comments are identical, we label each sample as positive (old comment is obsolete) or negative (old comment is not obsolete). Then, a detector named OCD and an updater named CUP are trained using the flattened data. Both of them take as input the old code, new code and old comment. The label is used to guide the training of OCD, while CUP only uses positive samples for training and considers the new comment as ground truth. Finally, given a code change and its associated old comment, CUP^2 first leverages the trained OCD to predict whether the old comment needs to be updated. If the answer is yes, the trained CUP will be used to generate a new comment to replace the old one.

OCD and CUP use the same encoder architecture to learn the representation of code changes and old comments, but they connect their encoders with different output components to perform prediction and generation, respectively. They are also trained separately to focus on different aspects of code changes and comments.

In this section, we first describe the data flattening phase. Then we elaborate on the two neural network models in CUP^2, i.e., OCD and CUP, by presenting the common encoder and the different output components used by them.

4.1 Data Flattening

In this phase, we convert code changes and comments into sequences so that they can be processed by OCD and CUP.

4.1.1 Tokenization

For comments, we first tokenize them by spaces and punctuation marks. Spaces are removed while punctuation marks are kept. Then, compound words, which refer to the tokens constructed by concatenating multiple vocabulary words according to camel or snake conventions, are split into multiple tokens to reduce OOV words. After that, if two adjacent tokens are not split by space, we insert a special token "$\langle con \rangle$" between them to mark they are concatenated.

As for code changes, each of them is composed of an old code snippet and a new code snippet. The two snippets are first tokenized into words using a lexer. Inner comments and white spaces are removed. Each identifier is further tokenized into tokens based on camel casing and snake casing, and "$\langle con \rangle$" is also inserted to join the tokens. String literals are tokenized like comments.

The key issue in software artifact tokenization is how to deal with compound words. In the literature, the common ways include: not changing compound words [19], splitting them [17] and adding a special symbol "$\langle /t \rangle$" at the end of each token before splitting [45] (e.g., "inputBuffer" $\rightarrow$ "inputBuffer$\langle /t \rangle$" $\rightarrow$ "input Buffer$\langle /t \rangle$"). However, the first way cannot reduce OOV words. The second way may lose format information, i.e., a token sequence may fail to be recovered to its original sentence. The third way cannot handle the situation where a subtoken of a compound word is generated as an independent token. For example, "input" cannot be generated as an independent word if it is copied from "inputBuffer", since it does not end with "$\langle /t \rangle$". Compared to these methods, our tokenizer can be regarded as "asking" the neural model to also learn format information by inserting "$\langle con \rangle$" to mark concatenation. Simple is it, it can effectively keep comment format consistent. Also, to preserve format information, we do not lowercase tokens in both code and comments.

4.1.2 Code Change Representation

After tokenization, each code change is converted to two token sequences. We can simply use two encoders to encode them, which, however, makes it hard to capture fine-grained modifications between them. To better represent each code change, we first align its two code snippets at the token level using a diff tool and some heuristics, and then construct an edit sequence based on the alignment, similar to [46], as its representation, as shown in Figure 2. Each element in an edit sequence is a triple $\langle t_i, t'_i, a_i \rangle$ and is named as an *edit*. t_i is a token in the old code and t'_i is a token in the new code. a_i is the edit action that converts t_i to t'_i, which can be *insert*, *delete*, *equal* or *replace*. If a_i is *insert* (*delete*), t_i (t'_i) will be the empty token $\emptyset$. Such edit sequences can not only preserve the information of the old code and the new code, but also highlight the fine-grained changes between them.

It is worth mentioning that aligning two code snippets at the token level is not so straightforward even with a diff tool. The core task of this step can be described as: given two sequences of elements (e.g., words or tokens), get their proper element-level alignment. To tackle this challenge, we use a Python diff tool named SequenceMatcher. Unfortunately, the alignment computed by SequenceMatcher is at the subsequence level instead of element level. Therefore we further use some heuristics to refine such subsequence-level alignment. For a subsequence match marked as *insert*, *delete* or *equal*, converting it into one or more element matches is straightforward. For a *replace* subsequence match, the tokens between the two subsequences are all different, and we leverage the procedure presented in Algorithm 1 to convert it into element match(es). Specifically, given subsequence $A = [A_1, \cdots, A_i]$ and subsequence $B = [B_1, \cdots, B_j]$ matched by SequenceMatcher as *replace*, we first compute the similarity s_1 between A_1 and B_1 and the similarity s_n between A_i and B_j using the *quick_ratio* method provided by SequenceMatcher (Lines 3-4). i and j are the lengths of A and B, and they may not be equal. s_n refers to the similarity

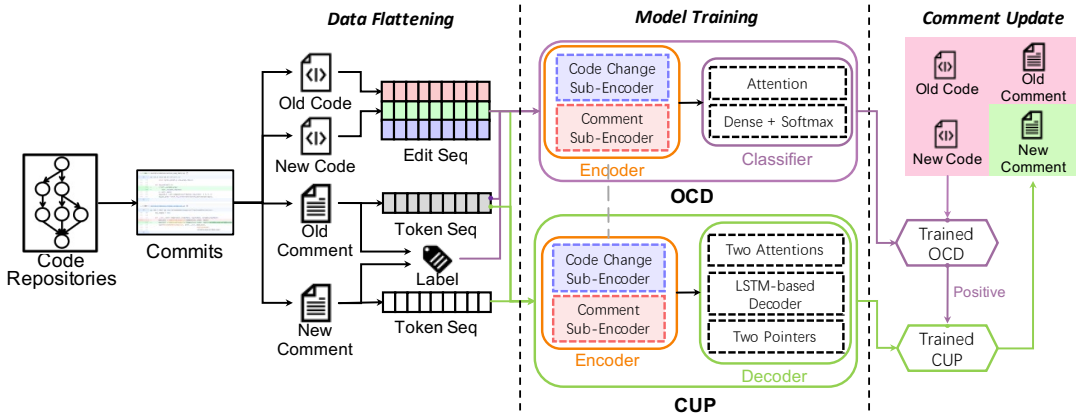

Fig. 1. The overall framework of CUP[2]

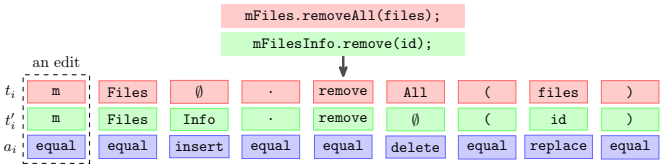

Fig. 2. Converting a code change to an edit sequence.

Algorithm 1: Compute the element-level alignment between two subsequences matched as *replace*

input : Two subsequences $A = [A_1, \cdots, A_i]$ and $B = [B_1, \cdots, B_j]$ where $A \cap B = \emptyset$
output: The alignment $align$ between A and B

1 $align \leftarrow []$;
2 $delta \leftarrow [\text{“}\text{”}] * |i - j|$;
3 $s_1 \leftarrow \text{similarity}(A_1, B_1)$;
4 $s_n \leftarrow \text{similarity}(A_i, B_j)$;
5 **if** $s_1 \geq s_n$ **then**
6 **if** $i > j$ **then**
7 $B \leftarrow B + delta$;
8 **else**
9 $A \leftarrow A + delta$;
10 **else**
11 **if** $i > j$ **then**
12 $B \leftarrow delta + B$;
13 **else**
14 $A \leftarrow delta + A$;
15 **for** $idx \leftarrow 1$ **to** $max(i, j)$ **do**
16 **if** $A_{idx} =$ “” **then**
17 align.append($[A_{idx}, B_{idx}, \text{“}insert\text{”}]$);
18 **else if** $B_{idx} =$ “” **then**
19 align.append($[A_{idx}, B_{idx}, \text{“}delete\text{”}]$);
20 **else**
21 align.append($[A_{idx}, B_{idx}, \text{“}replace\text{”}]$);

between the last elements of A and B. If $s_1 \geq s_n$, we align A and B from the beginning to the end to get the element-level alignment, and if $s_n > s_1$, we compute the alignment from the opposite direction (Lines 5-21).

With the procedure mentioned above, given a code change, we can convert it into a token-level alignment as follows: 1) We tokenize its two code snippets into two word sequences using a lexer, and compute its word-level alignment using the procedure. 2) Each computed word match is converted into two token sequences using the tokenizer described in Section 4.1.1, and the token-level alignment of this word match can also be computed using the aforementioned procedure. 3) All the token matches computed from the word-level alignment are concatenated as the token-level alignment of the code change.

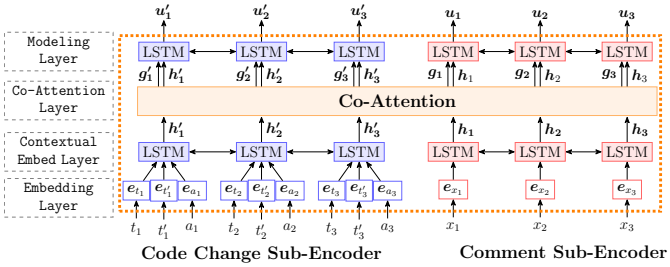

Fig. 3. The encoder architecture used by OCD and CUP.

4.2 Encoder

OCD and CUP use the same encoder architecture to learn the representations of code changes and old comments, which is presented in Figure 3. The encoder takes as input an edit sequence $\boldsymbol{E} = [\langle t_1, t'_1, a_1\rangle, \cdots, \langle |t_n, t'_n, a_n\rangle]$ and an old comment $\boldsymbol{x} = [x_1, \cdots, x_{|\boldsymbol{x}|}]$, and aims to convert each edit and each comment token into feature vectors that capture enough contextual information for obsolete comment prediction or new comment generation. In detail, the encoder consists of two distinct sub-encoders, i.e., Code Change Sub-Encoder and Comment Sub-Encoder, to encode the edit sequence and the old comment, respectively. The two sub-encoders are nearly the same in structure. Each of them is composed of four ordered layers: an embedding layer, a contextual embed layer, a co-attention layer and a modeling layer.

4.2.1 The Embedding Layer

This layer is responsible for mapping the three kinds of tokens, i.e., code tokens, comment tokens, and edit actions,

into embeddings. There are only four edit actions, so we randomly initialize an embedding matrix for them and update it during training. For code and comment tokens, we first build a unified vocabulary from all training code and comment tokens. Then we use a pre-trained fastText model [47] to obtain the word embedding of each token. Instead of two distinct vocabularies for code and comments, we prefer a unified one. Because with its help, we can map code tokens and comment tokens into one shared vector space, which can ensure the same tokens in code and comments have the same embeddings and ease the capture of code-comment references. Pre-trained word embeddings are used for providing accurate syntactic and semantic information. In addition, we choose fastText instead of other pre-trained models because the word embeddings learned by fastText contain subtoken information and it can effectively map subtokens (e.g., the tokens split from compound words) into embeddings, which are very suitable for this task.

4.2.2 The Contextual Embed Layer

For each encoder, we place a distinct Bi-LSTM (Bidirectional LSTM) on the top of the embedding layer to model the temporal interactions between edits (comment tokens) and represent each edit (comment token) as a contextual vector. For Code Change Sub-Encoder, the three embeddings, i.e., $\boldsymbol{e}_{t_i}$, $\boldsymbol{e}_{t'_i}$ and $\boldsymbol{e}_{a_i}$, of each edit $\boldsymbol{E_i}$ are first concatenated horizontally, and then input to the Bi-LSTM, as follows:

$$\boldsymbol{h'_i} = \text{Bi-LSTM}(\boldsymbol{h'_{i-1}}, \boldsymbol{h'_{i+1}}, [\boldsymbol{e}_{t_i}; \boldsymbol{e}_{t'_i}; \boldsymbol{e}_{a_i}])$$

where $\boldsymbol{h'_i}$ is the contextual vector of this edit. Comment Sub-Encoder computes the contextual vector $\boldsymbol{h_i}$ of each comment token x_i in a similar way:

$$\boldsymbol{h_i} = \text{Bi-LSTM}(\boldsymbol{h_{i-1}}, \boldsymbol{h_{i+1}}, \boldsymbol{e}_{x_i})$$

where $\boldsymbol{e}_{x_i}$ is the embedding of x_i. For convenience, the contextual vectors of the old comment and the code change can be stacked as matrices $\boldsymbol{H} \in \mathbb{R}^{2d \times |\boldsymbol{x}|}$ and $\boldsymbol{H'} \in \mathbb{R}^{2d \times n}$, respectively.

4.2.3 The Co-Attention Layer

So far, the code change and the old comment are represented independently. However, to capture relationships between them, it is necessary to link and fuse their information. This layer is used to address this need and is shared by the two sub-encoders. It takes as input the contextual vectors, i.e., $\boldsymbol{H}$ and $\boldsymbol{H'}$, and outputs a comment-aware (edit-aware) feature vector for each edit (comment token) along with the original contextual vector of this edit (comment token) to the consequent layer.

In detail, each X-aware feature vector is indeed a context vector computed by the dot-product attention mechanism [48]. Formally, the edit-aware feature vector $\boldsymbol{g_i}$ of comment token x_i is calculated by:

$$\boldsymbol{g_i} = \boldsymbol{H'}\boldsymbol{\beta_i} \tag{2}$$

$$\boldsymbol{\beta_i} = \text{softmax}(\boldsymbol{H'}^{\top}\boldsymbol{W_\beta}\boldsymbol{h_i}) \tag{3}$$

$\boldsymbol{\beta_i}$ is the attention weights of x_i on all edits and measures how important each edit is with respect to x_i. $\boldsymbol{W_\beta} \in \mathbb{R}^{2d \times 2d}$ is the trainable parameters. The comment-aware feature vector $\boldsymbol{g'_i}$ of edit $\boldsymbol{E_i}$ is computed in nearly the same way except that the attentions are derived oppositely, i.e., from edits to comment tokens, as follows:

$$\boldsymbol{g'_i} = \boldsymbol{H}\boldsymbol{\beta'_i}$$

$$\boldsymbol{\beta'_i} = \text{softmax}(\boldsymbol{H}^{\top}\boldsymbol{W_\beta}^{\top}\boldsymbol{h'_i})$$

We can see that $\boldsymbol{g_i}$ signifies and captures the information related to comment token x_i from the whole code change. Meanwhile, $\boldsymbol{g'_i}$ highlights and keeps the information related to edit $\boldsymbol{E_i}$ from the whole old comment. These X-aware feature vectors provide a foundation for capturing relationships between code and comments.

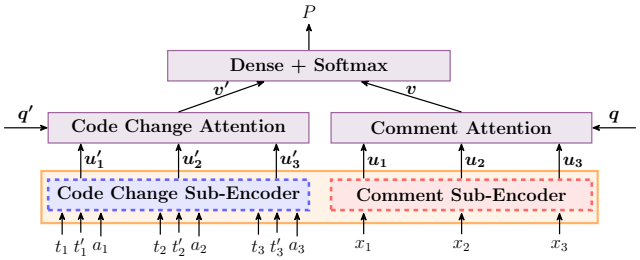

Fig. 4. The architecture of OCD

4.2.4 The Modeling Layer

This layer produces the final representation of each edit (comment token) based on its contextual vector and comment-aware (edit-aware) feature vector. The two sub-encoders use two distinct Bi-LSTMs to learn such representations. In detail, given a comment token x_i, its final representation $\boldsymbol{u_i}$ is calculated as follows:

$$\boldsymbol{u_i} = \text{Bi-LSTM}(\text{u}_{\boldsymbol{i-1}}, \boldsymbol{u_{i+1}}, [\boldsymbol{g_i}; \boldsymbol{h_i}])$$

The final representation $\boldsymbol{u'_i}$ of an edit $\boldsymbol{E_i}$ is calculated similarly:

$$\boldsymbol{u'_i} = \text{Bi-LSTM}(\text{u}'_{\boldsymbol{i-1}}, \boldsymbol{u'_{i+1}}, [\boldsymbol{g'_i}; \boldsymbol{h'_i}])$$

$\boldsymbol{u_i}$ ($\boldsymbol{u'_i}$) is expected to contain the contextual information of x_i ($\boldsymbol{E_i}$) with respect to both the code change and the old comment. For convenience, we refer to the stacked matrices of all $\boldsymbol{u_i}$ and all $\boldsymbol{u'_i}$ as $\boldsymbol{U} \in \mathbb{R}^{2d \times |\boldsymbol{x}|}$ and $\boldsymbol{U'} \in \mathbb{R}^{2d \times n}$, respectively.

4.3 Obsolete Comment Detector (OCD)

OCD in CUP^2 is responsible for predicting obsolete comments. It takes as input a code change and an associated old comment, and outputs the probability that this comment should be updated with this code change. Figure 4 presents the architecture of OCD. The encoder mentioned above is first used by OCD to encode the code change and the comment into sequences of feature vectors, i.e., $\boldsymbol{U}$ and $\boldsymbol{U'}$, simultaneously. Then, the dot-product attention mechanism is adopted to compute a single feature vector for the whole old comment and the whole code change, respectively:

$$\boldsymbol{v} = \boldsymbol{U}\boldsymbol{\omega}$$

$$\boldsymbol{\omega} = \text{softmax}(\boldsymbol{U}^{\top}\boldsymbol{W_\omega}\boldsymbol{q})$$

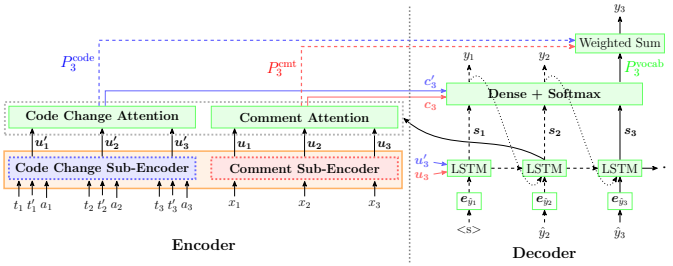

Fig. 5. The architecture of CUP

$$\boldsymbol{v'} = \boldsymbol{U'}\boldsymbol{\omega'}$$

$$\boldsymbol{\omega'} = \text{softmax}(\boldsymbol{U'}^{\top}\boldsymbol{W_{\omega'}}\boldsymbol{q'})$$

$\boldsymbol{\omega}$ and $\boldsymbol{\omega'}$ are attention weights, $\boldsymbol{q} \in \mathbb{R}^{2d}$ and $\boldsymbol{q'} \in \mathbb{R}^{2d}$ are learnable query vectors used to guide the "attentions" of $\boldsymbol{v}$ and $\boldsymbol{v'}$ to $\boldsymbol{U}$ and $\boldsymbol{U'}$. $\boldsymbol{W_{\omega}} \in \mathbb{R}^{2d\times 2d}$ and $\boldsymbol{W_{\omega}} \in \mathbb{R}^{2d\times 2d}$ are trainable parameters. We expect such attention mechanism can learn to effectively focus on and select the information that is important for obsolete comment prediction.

$\boldsymbol{v}$ and $\boldsymbol{v'}$ are then concatenated and passed through two linear layers with activation to calculate the probability of each label, as follows:

$$\boldsymbol{v_o} = \tanh(\boldsymbol{V_b}[\boldsymbol{v}; \boldsymbol{v'}]])$$

$$P = \text{softmax}(\boldsymbol{V'_b}\boldsymbol{v_o} + \boldsymbol{b})$$

where P is the probability distribution. $\boldsymbol{V_b} \in \mathbb{R}^{d\times 4d}$, $\boldsymbol{V'_b} \in \mathbb{R}^{2\times d}$ and $\boldsymbol{b}$ are trainable weights or biases.

4.4 Obsolete Comment Updater (CUP)

After prediction, CUP[2] uses CUP to generate the new version of each predicted obsolete comment with respect to the corresponding code change. CUP also leverages the aforementioned encoder to learn the representations of code changes and comments. But different from OCD, the output component of CUP is an LSTM-based decoder responsible for the generation of new comments.

Figure 5 illustrates the architecture of CUP and how a comment token is generated using the decoder. With the feature matrices $\boldsymbol{U}$ and $\boldsymbol{U'}$ output by the encoder as input, the decoder of CUP produces the new comment by sequentially generating its tokens. The initial state $\boldsymbol{s_0}$ of the decoder's LSTM is constructed by concatenating the last feature vectors of $\boldsymbol{U}$ and $\boldsymbol{U'}$. At each decoding step j, the input $\hat{y}_j$ is first mapped into an embedding $\boldsymbol{e_{\hat{y}_j}}$ using Comment Sub-Encoder's embedding layer. $\hat{y}_j$ is the previous reference token when training or the previously generated token when testing. Then, the hidden state $\boldsymbol{s_j}$ of this step is computed based on $\boldsymbol{e_{\hat{y}_j}}$, the previous hidden state $\boldsymbol{s_{j-1}}$ and the previous output vector $\boldsymbol{o_{j-1}}$ (computed by Equation 4), as follows:

$$\boldsymbol{s_j} = \text{LSTM}(\boldsymbol{s_{j-1}}, [\boldsymbol{e}_{\hat{y}_j}; \boldsymbol{o_{j-1}}])$$

The decoder also adopts the dot-product attention mechanism, which derives a context vector at each time step as the representation of the encoder's input. There are two sub-encoders, so the decoder calculates two context vectors, i.e., $\boldsymbol{c_j}$ from the old comment and $\boldsymbol{c'_j}$ from the code change, following Equation 2 and 3.

Then, $\boldsymbol{c_j}$, $\boldsymbol{c'_j}$ and $\boldsymbol{s_j}$ are concatenated to calculate an output vector $\boldsymbol{o_j} \in \mathbb{R}^l$ and a vocabulary distribution P_j^{vocba}:

$$\begin{aligned} \boldsymbol{o_j} &= \tanh(\boldsymbol{V_c}[\boldsymbol{c_j}; \boldsymbol{c'_j}; \boldsymbol{s_j}]) \\ P_j^{\text{vocab}} &= \text{softmax}(\boldsymbol{V'_c}\boldsymbol{o_j}) \end{aligned} \tag{4}$$

$\boldsymbol{V_c} \in \mathbb{R}^{l\times(4d+l)}$ and $\boldsymbol{V'_c} \in \mathbb{R}^{v\times l}$ are learnable parameters and v is the size of the unified vocabulary. P_j^{vocab} can be directly used to generate the target token. For example, we can choose the token with the highest probability as the output of time step j.

Out-of-vocabulary (OOV) words are pervasive in software artifacts. The decoder cannot generate OOV words if it only chooses tokens from a fixed vocabulary. We observed that an OOV word in a new comment usually can be found in its corresponding old comment and/or new code. Therefore, we also adopt the pointer generator [49] to alleviate the OOV problem following Liu et al. [19]. In addition, by copying from the old comment, the pointer generator can ease the generation of the new comment and help keep the format of the old and new comments consistent. Specifically, two pointer generators are used to copy tokens from the old comment and the new code, respectively:

$$P_j^{\text{cmt}}(y_j) = \sum_{k:x_k=y_j} \alpha_{jk}$$

$$P_j^{\text{code}}(y_j) = \sum_{k:t'_k=y_j} \alpha'_{jk}$$

$P_j^{\text{cmt}}(y_j)$ and $P_j^{\text{code}}(y_j)$ are the probabilities of copying y_j from the old comment and the new code. α_{jk} and α'_{jk} are the attention weights of x_k and $\boldsymbol{E_k}$ with respect to time step j, and are calculated with the context vectors $\boldsymbol{c_j}$ and $\boldsymbol{c'_j}$.

At last, the conditional probability of producing y_j at time step j is computed by combining P_j^{vocab}, P_j^{cmt} and P_j^{code}:

$$\begin{aligned} p(y_j|\boldsymbol{y}_{<j}, \boldsymbol{x}, \boldsymbol{E}) = \gamma_j P_j^{\text{vocab}}(y_j) + \\ (1-\gamma_j)(\theta_j P_j^{\text{cmt}}(y_j) + (1-\theta_j)P_j^{\text{code}}(y_j)) \end{aligned} \tag{5}$$

γ_j and θ_j measure the probabilities of generating y_j by selecting from the vocabulary and copying from the old comment, respectively. Each of them is modeled by a single-layer feed-forward neural network jointly trained with the decoder, as follows:

$$\gamma_j = \sigma(\boldsymbol{w}_{\gamma}^{\top}\boldsymbol{o_j} + b_{\gamma})$$

$$\theta_j = \sigma(\boldsymbol{w}_{\theta}^{\top}\boldsymbol{o_j} + b_{\theta})$$

where $\boldsymbol{w}_{\gamma}, \boldsymbol{w}_{\theta} \in \mathbb{R}^l$ and $b_{\gamma}, b_{\theta} \in \mathbb{R}$ are trainable parameters, and σ is the sigmoid function.

5 Data Preparation

We build a large-scale dataset for training and evaluating CUP[2]. This dataset is constructed from Java programs. However, our approach is language-agnostic and we believe it can be easily adapted for other languages. We concentrate on the modified methods in code changes and their header comments (method comments), because 1) Java methods

can be precisely associated with their header comments, while it is non-trivial to accurately link comments and code of other granularity, e.g., a statement and 2) Method comments are an important type of comments. They are often referred to by developers for program comprehension, and can be used to construct API reference documentation [50]. Also, our approach detects and updates obsolete comments at the sentence level, i.e., processes one comment sentence at a time. The reasons behind this choice are: 1) it is relatively easy to recognize patterns at a small but coherent granularity [46] and 2) a method comment with multiple sentences can also be processed and updated iteratively. In addition, our approach focuses on updating comments, and we regard deleting, adding and rewriting comments as separate problems. For convenience, in this section we use *comment* to refer to a comment sentence and *doc* for a whole method comment.

This section describes how we extract method-doc change instances, i.e., ⟨*old code*, *new code*, *old doc*, *new doc*⟩, from code repositories, how we convert qualified instances into method-comment change samples, i.e., ⟨*old code*, *new code*, *old comment*, *new comment*⟩, and how we build the datasets for training OCD and CUP and evaluating CUP^2, respectively. Note that *old doc* and *new doc* above can be identical; similarly *old comment* and *new comment* above can also be identical.

5.1 Data Collection

Wen et al. [30] collected 1,500 Java repositories from GitHub for studying code-comment inconsistencies. All the repositories were selected based on a rigorous procedure, have no less than 500 commits, and were manually verified by Wen et al. to be popular engineered projects. We also collect data from these projects. In detail, we first cloned the 1,500 repositories from GitHub. However, we found two repositories, i.e., *pig4cloud/pig* and *wyouflf/xUtils*, had been removed from GitHub and two other repositories, i.e., *liferay/liferay-portal* and *JetBrains/MPS*, were too large to be cloned in reasonable time. Therefore, 1,496 repositories were successfully cloned. Then, we constructed method-doc change instances by extracting modified methods and their corresponding docs from each non-merge commit of every repository. Methods and docs were associated using JavaParser [51]. We obtained 6,106K method-doc change instances after filtering out the instances with empty old doc or empty new doc. Different from our previous work [21], we do not remove the instances where the old and new docs are identical, because such instances are needed for extracting negative method-comment change samples, which are necessary for training and evaluating OCD.

5.2 Modified Method Extraction

It is non-trivial to extract modified methods from a commit, since developers may change method signatures. To do this, we first leveraged GumTree [52] to calculate method mappings between two revisions. Then, based on such mappings, we compared the ASTs of each old method and its new version to identify and extract modified methods. Comments were ignored for AST comparisons.

However, GumTree is not designed for matching methods and we found that for short methods and methods with similar bodies, the method mappings extracted by GumTree are not always accurate. To alleviate this problem, we customized GumTree's matching algorithm to better extract method mappings. In detail, GumTree's matching algorithm takes two trees T_1 and T_2 as input and contains two ordered phases: the top-down phase and the bottom-up phase. The top-down phase matches isomorphic subtrees between the two trees and the bottom-up phase tries to find additional mappings in a bottom-up way. We customized GumTree by adding an additional phase named method-matching phase *between* the two phases.

This method-matching phase is based on our observation that if method m_i in T_1 and method m_j in T_2 have the same signature, they usually should be matched. In addition, if m_i's signature is different from that of m_j but they have the same name and no other methods in both T_1 and T_2 use this name, it is very likely that m_j is modified from m_i. Specifically, this phase first collects unmatched *MethodDeclaration* nodes $\mathcal{M}_1$ and $\mathcal{M}_2$ from T_1 and T_2, respectively. Then, for each method m_i in $\mathcal{M}_1$, if only one method m_j in $\mathcal{M}_2$ has the same signature as it, m_i and m_j are matched and removed from $\mathcal{M}_1$ and $\mathcal{M}_2$. After checking method signatures and updating $\mathcal{M}_1$ and $\mathcal{M}_2$, this phase continues to match the remaining methods in $\mathcal{M}_1$ and $\mathcal{M}_2$ with respect to method names in a similar way.

It took over 290 hours to extract modified methods from the 1,496 repositories using 40 cores of Intel Xeon 2.7GHz CPU.

5.3 Data Preprocessing

After obtained the 6,106K method-doc change instances, we preprocessed them as follows:

5.3.1 Filter Out Unqualified Method-Doc Change Instances

This step aims to reduce unrelated information in docs and filter out the instances of which the docs are not method descriptions or the methods are mismatched. We observed that if a doc is a line comment (i.e., a comment starting with "//") instead of a Javadoc or a block comment, it is usually a commented Java annotation (e.g., "//@Override") instead of a method description. So, we first removed the instances with line comments as docs. A doc can consist of a free-form description section and a tag section. The description section usually contains "what", "why" and "how" information of the associated method, while the tag section describes the parameters and return value of the method. Compared to the description section, the tag section is more formal and structured, and can be well handled using rule-based methods. Therefore, we focused on the description section and deleted the tag section in each doc. Next, "@inheritDoc", code snippets and HTML tags were removed from each doc to reduce noise in comments. Then, the docs containing the phrase "(non-Javadoc)" were filtered out, because we noticed that such docs always provide no description of their associated methods and only consist of one or more "@see" tags referring to other methods. The docs with non-ASCII characters were also deleted. After these operations, empty

docs may appear and we further removed the instances with empty docs. Finally, according to our inspection of the method mappings computed by GumTree, GumTree often produces inaccurate mappings for abstract methods. Therefore, the instances containing abstract methods were also deleted to reduce method mismatching. After this step, we obtained 4,357K qualified method-doc change instances.

5.3.2 Construct Method-Comment Change Samples

A doc may contain multiple sentences. This step is responsible for further processing docs and matching sentences between each old doc and its new doc to construct method-comment change samples. Before sentence matching, we first replaced emails, URLs, references (e.g., "#123") and versions (e.g., "1.2.3") in docs with "EMAIL", "URL", "REF" and "VERSION", respectively, to reduce noise. Next, we split each doc into sentences using NLTK [53], removed the sentences with only punctuation marks, and tokenized the remaining sentences using the tokenizer described in Section 4.1.1. Then, given a pair of docs, we calculated the word-level Levenshtein distance [54], which is the minimum word edits (insertions, deletions and substitutions) required to change a sentence into the other, between each old sentence and each new sentence, and constructed a distance matrix. Based on this matrix, the old and new sentences are matched in a best-fit way. If the distance of a matched pair is larger than the old sentence's length, this pair should be regarded as a rewrite instead of an update. We notice that some old sentences are very short, e.g., "get Element", and some updates on such short old sentences, e.g., updating "get Element" to "get a list of Element", may be mistakenly regarded as rewrites. We used a threshold to mitigate such mistakes, and regarded the matched pairs of which the distances are larger than both the corresponding old sentences' lengths and the threshold as rewrites. After manually inspecting 100 randomly selected matched pairs with non-zero distances, we found that when the threshold is set to 5, there was no mistakenly identified rewrite pair. Therefore, we identified rewrite pairs with 5 as the threshold and filtered out rewrite pairs after sentence matching. Finally, each remaining matched pair was used to construct a method-comment change sample, i.e., $\langle old\ code, new\ code, old\ comment, new\ comment \rangle$. We can see that one method-doc change instance can be split into multiple method-comment change samples, which share the code change but have their own sentence pairs. As a result, we constructed 5,352K method-comment change samples, which belong to 3,282K method-doc change instances. If the old and new comments of a sample are different after removing punctuations, we regard the old comment as an obsolete comment and label this sample as a **positive sample**. Otherwise, this sample is labeled as a **negative sample**.

5.3.3 Set Max Length and Max Distance

Due to the limited memory of GPU and to reduce the training time, we set the max lengths of code edit sequences, old comments, and new comments to be 500, 50, and 50, respectively. More than 95% of the processed comments have less than 50 tokens. In addition, a comment change is very likely to be a rewrite instead of an update if the absolute or relative edit distance between the old and new comments is large. The relative edit distance is defined as the absolute distance divided by the old comment's length. We find that the absolute and relative edit distances of 80% positive samples are no more than 12 and 0.67, respectively. To reduce comment-rewrite samples, we filtered out the samples of which the absolute or relative distances is larger than 12 or 0.67. At last, we obtained 4,206K method-comment co-change samples, which come from 2,693K method-doc co-change instances.

5.4 Data Splitting

The 4,206K method-comment change samples are extracted from 707K commits. Since developers may perform systematic or recurring code changes in one commit [55], [56], a commit may contain duplicate change samples. So, before splitting the data, for every duplicate sample set within each commit, we only kept one sample and removed the others in it to avoid duplicate samples inflate the performance of the evaluated approaches. For each project, we sorted its commits in the ascending order of commit creation time, put the first 80% commits into the training set, shuffled the remaining 20% commits, and evenly split them into the validation and test set. In this way, we ensure all changes in the training set happened before those in the validation and test sets. We also noticed that git operations like "cherry-pick", "rebase" and "squash" can introduce duplicate samples among different commits. Therefore, after splitting, duplicate samples between the test (validation) and training sets were also filtered out by us. As a result, our final training, validation and test sets consist of 3,194K, 437K and 454K method-comment change samples, respectively.

Such training and validation sets are used to train and validate OCD, and the evaluations of CUP[2] and OCD are conducted on the test set. For convenience, we refer to this dataset as **Whole** dataset. Different from OCD, CUP only focuses on the positive samples, i.e., the samples where the old comments require updates. To learn the update patterns of comments, we need to train and evaluate CUP on a dataset with only positive samples. Therefore, we extracted the positive samples from the training, validation and test sets of the Whole dataset, respectively, to construct a sub-dataset named **Update** dataset. The training, validation and test sets of the Update dataset contain 85.4K, 9.8K and 9.6K method-comment change samples, respectively.

Please note that the Whole dataset is highly imbalanced and the ratio of the positive samples to the negative samples in it is about 1/38. The possible reasons for this phenomenon are: 1) comment updates are less frequent than method changes; 2) the Whole dataset targets at comment sentences and when developers update a comment, usually only part of its sentences are updated; 3) there may exist some mislabeled negative samples, i.e., some samples actually need to be updated but developers have not update them yet.

Another thing worth mentioning is the Update dataset and the dataset built in our previous work [21], namely the **Original** dataset, are split in different ways. The Update dataset is split according to the commit creation time of all change samples, while the Original dataset only considers

the positive samples. Therefore, the training, validation and test sets of the Update and the Original datasets consist of different sets of samples, respectively.

6 Evaluation Setup

In this section, we first present the baselines and the evaluation metrics used to assess the performance of CUP^2 and its two components. Then, we describe our experiment settings used for evaluation.

6.1 Baselines

To evaluate the effectiveness of CUP^2 as well as its two components, i.e., OCD and CUP, we choose the following baselines:

6.1.1 Baselines for CUP^2

There is little work focusing on combining the detection and update of obsolete comments. The tool that is most related to CUP^2 is Fraco [15], so we use it as the baseline:

Fraco is a rule-based tool proposed by Ratol and Robillard to detect fragile comments with respect to rename refactorings and is shown to perform better than Eclipse's refactoring tool. Although the paper proposing Fraco only presents Fraco's ability to detect fragile comments and does not claim that Fraco can update fragile comments with rename refactorings, we find the implementation of Fraco provides a quick-fix feature to fix detected fragile comments. When developers conduct a rename refactoring, Fraco will be triggered to identify the references between comment phrases and the renamed identifier. If any fragile comment phrase is detected, the quick-fix feature can be used to automatically replace such phrase with the new identifier name based on heuristic rules. We manually extract the detection algorithm and the quick-fix feature from Fraco's source code, wrap them as an offline tool, and use this tool to conduct experiments. Given a code change and a corresponding old comment, the offline Fraco first leverages RefactoringMiner [57] to detect rename refactorings from the code change. Then, for each detected rename refactoring, it uses Fraco's detection algorithm to identify fragile comment phrases in the old comment. Finally, Fraco's quick-fix feature is applied to fix detected fragile phrases.

6.1.2 Baselines for OCD

To assess the effectiveness of OCD, we compare it with two baselines, namely FracoDetector and RandomForest, for evaluation:

FracoDetector is the detection component of the offline Fraco mentioned above, and is used as a rule-based baseline for OCD. Specifically, given a code change and a corresponding old comment, it performs rename refactoring extraction and fragile phrase detection. If there is no rename refactoring extracted or no fragile comment phrase identified, FracoDetector predicts this sample as negative. Otherwise, this sample is regarded as positive.

RandomForest is a machine-learning-based approach proposed by Liu et al. [16] to detect outdated block or line comments during code changes. Specifically, Liu et al. manually crafted 64 features related to code, old comments and the relationship between them, and employed a random forest for prediction. Unfortunately, they did not provide the implementation of their approach. Therefore, we re-implement their approach using JavaParser [51], GumTree [52], RefactoringMiner [57] and scikit-learn [58] and use this re-implementation as a baseline for OCD. Different from Liu et al. [16], this work takes the change of a single method as input. Therefore, some features requiring the context beyond a single method's change are not suitable for this work and cannot be extracted from our dataset. Such features include 2 class-level code features (changes on class attributes, class attribute related), 4 refactoring features (extract method, inline method, encapsulate field, replace exception with test) and 1 class-level comment feature (the ratio of comment lines to the class). In addition, since this work focuses on method comments, two comment features, i.e., the ratios of comment lines to the method and to the code snippet, are the same and should be merged into one. Due to these considerations, our re-implementation uses the remaining 56 features instead of all 64 features and a random forest for obsolete comment detection. For convenience, we refer to our re-implementation as RandomForest.

6.1.3 Baselines for CUP

We use three baselines belonging to different types for evaluating CUP, i.e., Origin, FracoUpdater and NNUpdater.

Origin is a special baseline for CUP. Given a code change and a corresponding old comment requiring updates, it directly outputs the old comment as the result. By comparing CUP with Origin, we can know whether the comments generated by CUP are closer to the new comments than the old comments.

NNUpdater, short for Nearest-Neighbor-based comment Updater, is an IR-based baseline proposed by us. Like NNGen [59] for commit message generation, the hypothesis behind NNUpdater is that similar code changes may lead to similar or even the same comment changes. Given a test sample, i.e., a code change and its old comment requiring updates, NNUpdater first finds its most similar training sample and then reuses the new comment of the nearest neighbor as output. Specifically, to measure the similarity sim_{chg} between two code changes, NNUpdater converts each of them to a unified *diff* file, represents *diff* files as tf-idf vectors, and calculates the cosine similarity between such vectors. The similarity sim_{cmt} between two old comments are calculated in the same way. The final similarity sim between two samples is then defined as: $sim = \alpha \cdot sim_{\text{chg}} + (1 - \alpha) \cdot sim_{\text{com}}, 0 \leq \alpha \leq 1$. α is tuned using the validation set.

FracoUpdater is the update component of the offline Fraco mentioned in Section 6.1.1. Given a code change and a corresponding obsolete comment, it performs rename refactoring extraction, fragile phrase detection and fragile phrase fixes in order. For those samples where no rename refactoring is extracted or no fragile comment phrase is identified, FracoUpdater outputs the old comments as the results. FracoUpdater is used as a rule-based baseline.

6.1.4 Variants of OCD and CUP

In the encoder architecture used by both OCD and CUP, we adopt a co-attention mechanism, build a unified vocabulary, and use the fastText word embeddings to better capture the

relationships between code changes and old comments. In addition, the decoder of CUP leverages pointer generators to generate OOV words and preserve comment format. To understand the effects of these components on the performance of OCD and CUP, we construct several variants of OCD and CUP, namely OCD-co-attn, OCD-uni-vocab, OCD-fastText, CUP-co-attn, CUP-uni-vocab, CUP-fastText, CUP-pointer:

OCD-co-attn and CUP-co-attn remove the co-attention layer from their encoders and keeps other components the same as OCD and CUP, respectively.

OCD-uni-vocab and CUP-uni-vocab both use two distinct vocabularies, instead of a unified one, for code and comment tokens. Specifically, in the encoder, a vocabulary built from the code snippets in the training set, namely *code vocabulary*, is used to encode code tokens by the Code Change Sub-Encoder; a vocabulary built from the comments in the training set, namely *comment vocabulary*, is used to encode old comment tokens by the Comment Sub-Encoder. The decoder of CUP-uni-vocab also uses the *comment vocabulary* for generating comment tokens and encoding the tokens generated previously.

OCD-fastText and CUP-fastText do not use the fastText pre-trained word embeddings but learn word embeddings during training.

CUP-pointer removes the pointer generators used in the decoder and keeps other components consistent with CUP. In other words, it produces comment tokens only by selecting them from the vocabulary.

6.2 Evaluation Metrics

The metrics used to evaluate OCD, CUP and CUP^2 are as follows:

6.2.1 Evaluation Metrics for OCD

The detection of obsolete comments is a binary classification problem. OCD is actually a binary classifier. So we use **Precision**, **Recall** and **F1-Score**, which are well-known metrics for binary classification, to measure the performance of OCD.

6.2.2 Evaluation Metrics for CUP

Given a code-comment change sample, CUP aims at generating a new comment to replace the old comment based on the code change. We use **Accuracy**, **Recall@5**, **GLEU** [20], [25] and two evaluation metrics proposed by us for the JIT comment update task, namely Average Edit Distance (**AED**) and Relative Edit Distance (**RED**), to evaluate CUP and the corresponding baselines.

Accuracy and **Recall@5** are used to present to what extent a comment update approach can generate *correct comments*. We use *correct comments* to refer to the generated comments which are identical to the reference comments if we ignore the punctuation marks at the end of the comments. In detail, Accuracy is the percentage of the test samples where *correct comments* are generated on the first tries. Recall@5 is similar to Accuracy, but it allows an approach to generate 5 candidates for each test sample and if any generated candidate is a *correct comment*, it considers that this approach can successfully update this sample. For each test sample, CUP can generate multiple candidates through beam search. Specifically, on each decoding step, CUP keeps trace of the k most probable partial comments generated so far based on the probabilities computed by Equation 5. K is the beam size. Hence, CUP can produce 5 candidates at the end of decoding by setting k to 5. NNUpdater is able to produce 5 candidates from the 5 most similar training samples. However, Origin and FracoUpdater only generate one candidate for each sample, so their Recall@5 will be marked as "N/A".

GLEU is an automatic metric originally proposed to evaluate Grammatical Error Correction (GEC) systems in the natural language processing field. It measures how close a generated sentence ($\hat{\boldsymbol{y}}$) is to its reference ($\boldsymbol{y}$) with respect to its source sentence ($\boldsymbol{x}$) based on n-gram overlaps among these sentences. GLEU is more flexible than Accuracy and Recall@5 and is shown to highly correlate with human judgments on GEC tasks [25].

In our case, given a test sample, $\hat{\boldsymbol{y}}$ refers to the comment generated by a comment updater, $\boldsymbol{y}$ is the new comment written by developers, and $\boldsymbol{x}$ refers to the old comment of this sample. In detail, GLEU deviates from BLEU [60] and is also computed as the geometric mean of the modified n-gram precision scores (p_n), multiplied by a brevity penalty (BP), as follows:

$$\mathrm{GLEU}(\boldsymbol{x}, \boldsymbol{y}, \hat{\boldsymbol{y}}) = BP \cdot \exp(\sum_{n=1}^{N} w_n \log p_n) \tag{6}$$

$$BP = \begin{cases} 1, & \text{if } |\hat{\boldsymbol{y}}| > |\boldsymbol{y}| \\ e^{1-|\boldsymbol{y}|/|\hat{\boldsymbol{y}}|}, & \text{otherwise} \end{cases} \tag{7}$$

where w_n is the weight of p_n. To capture the quality of $\hat{\boldsymbol{y}}$ and highlight the differences between $\boldsymbol{x}$ and $\boldsymbol{y}$, p_n is defined to reward the n-gram overlaps between $\hat{\boldsymbol{y}}$ and $\boldsymbol{y}$ and penalize the n-grams in $\boldsymbol{x}$ that should have been updated but are not, as follows:

$$p_n = \frac{\sum\limits_{ng \in \{\hat{\boldsymbol{y}} \cap \boldsymbol{y}\}} \mathrm{Cnt}_{\hat{\boldsymbol{y}},\boldsymbol{y}}(ng) - \mathrm{Miss}_{\boldsymbol{x},\boldsymbol{y},\hat{\boldsymbol{y}}}(ng)}{\sum\limits_{ng \in \{\hat{\boldsymbol{y}}\}} \mathrm{Cnt}_{\hat{\boldsymbol{y}}}(ng)}$$

$$\mathrm{Miss}_{\boldsymbol{x},\boldsymbol{y},\hat{\boldsymbol{y}}}(ng) = \sum_{ng \in \{\hat{\boldsymbol{y}} \cap \boldsymbol{x}\}} \max[0, \mathrm{Cnt}_{\hat{\boldsymbol{y}},\boldsymbol{x}}(ng) - \mathrm{Cnt}_{\hat{\boldsymbol{y}},\boldsymbol{y}}(ng)]$$

where ng refers to an n-gram, $\mathrm{Cnt}_{\hat{\boldsymbol{y}}}(ng)$ denotes the count of ng in $\hat{\boldsymbol{y}}$ and $\mathrm{Cnt}_{\boldsymbol{a},\boldsymbol{b}}(ng)$ calculates the minimum count of ng in $\boldsymbol{a}$ and $\boldsymbol{b}$. In Equation 6, BP is used to penalize short generated comments and control for recall. According to the common practice in the GEC literature, w_n is set to $\frac{1}{N}$ and N is set to 4.

GLEU is computed at the corpus level and is usually reported as a percentage value between 0 and 100. The higher the GLEU score is, the closer the generated comments are to the references.

AED and **RED** measure the edits developers need to perform to perfectly update comments after using a JIT comment updater. AED is the average edit distance between the generated comments and their references. RED is similar to AED but measures the average of relative edit distances.

Formally, given a test set with M samples, a comment updater's AED and RED calculated as follows:

$$AED = \frac{1}{M}\sum_{k=1}^{M}\text{edit\_distance}(\hat{\boldsymbol{y}}^{(k)}, \boldsymbol{y}^{(k)})$$

$$RED = \frac{1}{M}\sum_{k=1}^{M}\frac{\text{edit\_distance}(\hat{\boldsymbol{y}}^{(k)}, \boldsymbol{y}^{(k)})}{\text{edit\_distance}(\boldsymbol{x}^{(k)}, \boldsymbol{y}^{(k)})}$$

where edit_distance is the word-level Levenshtein distance. $\hat{\boldsymbol{y}}^{(k)}$ refers to the comment generated for the k_{th} sample. AED and RED can measure and compare the performance of different comment update approaches in more detail. If an approach's RED is less than 1, we can expect that this approach can help developers understand where and how to update comments and reduce developers' edits on updating comments.

6.2.3 Evaluation Metrics for CUP^2

CUP^2 is the combination of OCD and CUP. We evaluate the performance of CUP^2 and its baseline in terms of all the evaluation metrics for OCD, i.e., **Precision**, **Recall** and **F1-Score**, and part of the evaluation metrics for CUP, including **AED** and **RED**. Accuracy, Recall@5 and GLEU are not used. Because the predictions of different obsolete comment detectors are usually different, which means the sets of test samples fed to the comment updaters of CUP^2 and its baseline can also be different. Under this situation, Accuracy, Recall@5 and GLEU would not produce fair estimates of the performance of comment updaters.

In addition, two other metrics named **#TPC/#TP** and **#FPC/#FP** proposed by us are reported. #TP and #FP refer to the numbers of true positive (TP) samples and false positive samples, respectively. #TPC refers to the number of TP samples that are correctly updated by an approach. #FPC denotes the number of FP samples that are not updated by an approach, i.e., the generated comments are the same as the corresponding old comments.

6.3 Experiment Setup

We use 300-dimensional word embeddings for edit actions, code tokens and comment tokens. The fastText model is pre-trained on Common Crawl and Wikipedia [61], and the pre-trained word embeddings are frozen during training. The hidden states of the Bi-LSTMs and the LSTMs are 256 and 512 dimensions (i.e., d=256 and l=512), respectively. All the LSTMs have only one layer. The dimensions of the query vectors used in OCD are set to 512, i.e., $2d$. The unified vocabularies used by OCD and CUP are built separately. For OCD, the vocabulary size is limited to 100,000 considering efficiency and the limited GPU memory. The vocabulary used by CUP only keeps the tokens appearing more than once, and its size turns out to be 44K.

OCD and CUP are trained separately and do not share their encoders. The training procedures of OCD and CUP are similar. They are both trained to minimize the cross entropy. Adam [62] with learning rate 0.001 is used as the optimizers, and the gradient norm is clipped by 5. The batch sizes are set to 192 and 24 for OCD and CUP, respectively. The size of the CUP model is bigger than that of the OCD model, and the largest batch sizes for OCD and CUP are approximately 384 and 64 under the limitation of our GPU memory. We trained OCD with batch sizes of [64, 128, 192, 256, 320, 384] and CUP using batch sizes from 8 to 64 with step size of 8, and found OCD performs best on the Whole validation set with batch size 192 and CUP achieves the best performance on the Update validation set with batch size 24. In addition, since OCD and CUP are trained separately, it is not necessary to make the ratio of CUP's batch size to OCD's batch size in line with the ratio of positive samples to all samples in the Whole dataset.

OCD and CUP are validated every 1,300 and 500 batches on their validation sets using cross entropy and perplexity, respectively. To improve the efficiency of training OCD, we randomly selected 100K samples from its validation set for conducting validation during training. For both OCD and CUP, the learning rate is decayed by a factor of 0.5 if the validation metric does not improve for 5 validations and we call this a trial. We stop training after 5 trials. The models with the best validation scores are used for evaluation. In addition, a dropout rate of 0.2 is used for all LSTM layers and the dense layer before computing P_j^{vocab} in CUP to avoid overfitting. We do not use dropout when training OCD since its training set is large enough. When testing CUP, beam search of width 5 is used to generate comments, and the maximum decoding step is set to 100. To reduce random error, OCD, CUP and CUP^2 are trained and evaluated 10 times and their average performance is reported as the evaluation results.

For RandomForest, to make our re-implementation reliable and our comparison fair, we keep its settings and hyper-parameters the same as the ones used by Liu et al. [16]. Specifically, RandomForest uses 100 Classification and Regression Trees (CART) as the base estimators. It considers 7 (the sqrt root of the number of all used features) features and adopts Gini function for searching the best split.

For NNUpdater, we tune its α on the validation set through grid search with 0.1 as the step size and find that the model with $\alpha = 0.7$ can achieve the best Accuracy. So, we set the α in NNUpdater to 0.7.

In addition, when comparing OCD, CUP and CUP^2 with their corresponding baselines and variants, Wilcoxon signed-rank tests [63] at the 95% confidence level and Cliff's delta [64] are used and computed respectively to check the significance and measure the effect sizes of the performance differences. Specifically, for each comparison, we conduct the statistical tests based on the 10 observations corresponding to the 10 times of model training and evaluation.

7 Evaluation Results

In this study, we want to investigate the following research questions (RQs):

RQ1: How effective is OCD on JIT obsolete comment detection?

RQ2: What are the impacts of the co-attention layer, the unified vocabulary and the fastText pre-trained word embeddings on OCD's performance?

RQ3: How effective is CUP on JIT obsolete comment update?

TABLE 2
Comparisons of OCD with two baselines

Approach	Precision	Recall	F1-Score
FracoDetector	21.7%	14.6%	17.4%
RandomForest	52.3%	10.5%	17.5%
OCD	**64.0%**	**17.1%**	**27.0%**
Imp.FracoDetector	194.9%****(L)	17.1%****(L)	55.2%****(L)
Imp.RandomForest	22.4%****(L)	62.9%****(L)	54.3%****(L)

**Imp indicates the improvement ratio of OCD over the baseline.
*, **, *** and **** refer that the corresponding p-value is less than 0.05, 0.01, 0.005 and 0.001.
N, **S**, **M** and **L** mean the corresponding effectiveness level is Negligible, Small, Medium and Large according to the Cliff's Delta.

TABLE 3
Detector test sample 1

Code Change:

```
- public String getData() {
-    return String.format("data:image/+ public byte[]
      getData() {
+    return data;
}
```

Old Comment: Gets a data URI for this image.

New Comment: Gets the raw data of the image.

Label: True

RQ4: What are the impacts of the co-attention layer, the unified vocabulary, the fastText pre-trained word embeddings and the pointer generators on CUP's performance?
RQ5: Can CUP[2] effectively detect and update obsolete comments given code changes?

7.1 RQ1: The effectiveness of OCD

Motivation: We want to know how well OCD can detect obsolete comments with code changes.
Approach: To answer this research question, we evaluate OCD, FracoDetector and RandomForest on the Whole dataset in terms of Precision, Recall and F1-Score.
Results: Table 2 presents the evaluation results. We can see that OCD achieves a Precision of 64.0% and a Recall of 17.1%. Although such performance is not perfect, we think it is reasonable because: 1) The precision of OCD is not bad; OCD outperforms the two baselines in terms of all metrics by at least 17.1%, and the improvements achieved by OCD over the baselines are all statistically significant with large effect sizes. These results indicate that compared to the baselines, OCD can detect more obsolete comments more accurately. 2) The detection of obsolete comments is not an easy task considering the difficulty of "understanding" code changes and comments, and the high-level of class imbalance of the Whole dataset. 3) OCD is the first attempt to automatically predict obsolete comments with code changes using carefully-crafted neural networks, which we hope can inspire and facilitate future research in this direction.

To figure out the possible reasons for OCD's better performance, we manually inspect some randomly selected test samples and review the predictions made by OCD and the baselines for them. According to the inspection, we owe OCD's better Recall to its two advantages: 1) Thanks to the specially designed neural network model, OCD can automatically handle more diverse code changes than the two baselines by learning from massive code-comment change data. In contrast, FracoDetector only focuses on code changes related to identifier renaming. The code features used by RandomForest only capture limited code change types, and many of them are coarse-grained. 2) Based on the special components, e.g., the co-attention layer, in the encoder, OCD is capable of capturing both lexical references and semantic relevance between code changes and comments. However, FracoDetector can hardly identify semantic code-comment relevance due to its dependence on token matching. Because of the coarse-grained code features, RandomForest is not so effective in capturing lexical references. Also, the relationship features used by RandomForest only include similarity scores between code and old comments as well as the distances of the similarities, which are also coarse-grained compared to the representations learned by the neural layers of OCD.

TABLE 4
Detector test sample 2

Code Change:

```
- public Response<SecretBase> updateSecretWithResponse(
     SecretBase secret, Context context) {
-    return client.updateSecretWithResponse(secret,
      context).block();
+ public Response<SecretProperties>
     updateSecretPropertiesWithResponse(SecretProperties
     secretProperties, Context context) {
+    return client.updateSecretPropertiesWithResponse(
      secretProperties, context).block();
 }
```

Old Comment: Gets the latest version of the secret, changes its expiry time and the updates the secret in the key vault.

New Comment: Gets the latest version of the secret, changes its expiry time and the updates the secret in the key vault.

Label: False

Table 3 presents a test sample. In this sample, the behavior of the method was modified, which made the old comment inconsistent with the code. Since there is no identifier renaming, FracoDetector fails to identify this obsolete comment. RandomForest also makes the wrong prediction. In contrast, OCD successfully predicts this sample to be positive, which demonstrates its effectiveness in capturing code-comment references.

The higher Precision of OCD reflects that OCD is more accurate than the two baselines in capturing references between code changes and comments. This is reasonable because the rules used by FracoDetector makes it difficult to precisely capture semantic code-comment relevance, and the coarse-grained features used by RandomForest hinder it from accurately capturing lexical references between code changes and comments. By comparison, the LSTM layers in OCD can help it learn tokens' contextual information. The special components, e.g., the co-attention layer, in OCD are useful for accurate token matching and effective capture of semantic code-comment relevance.

Table 4 presents a negative sample in the test set. In the commit where this sample is extracted, class "SecretBase" was renamed to "SecretProperties" and the method in this sample was modified to align with this renaming. FracoDetector is triggered by the renaming of parameter "secret" and matches this "secret" with the "secret" in the comment. Therefore, it predicts that this old comment needs

TABLE 5
Comparisons of OCD with three variants

Approach	Precision	Recall	F1-Score
OCD-co-attn	62.2%	16.0%	25.5%
OCD-uni-vocab	63.5%	16.6%	26.2%
OCD-fastText	61.5%	12.5%	20.7%
OCD	**64.0%**	**17.1%**	**27.0%**
Imp.OCD-co-attn	2.9%**(M)	6.9%⁻(M)	5.9%*(L)
Imp.OCD-uni-vocab	0.8%⁻(N)	3.0%⁻(S)	3.1%⁻(S)
Imp.OCD-fastText	4.1%*(L)	36.8%****(L)	30.4%****(L)

-, *, **, *** and **** refer that the corresponding p-value > 0.05, < 0.05, < 0.01, < 0.005, < 0.001
N, **S**, **M** and **L** mean the corresponding effectiveness level is Negligible, Small, Medium and Large according to the Cliff's Delta.

to be updated. RandomForest also predicts this sample to be positive. Only OCD distinguishes the meanings of the two "secret" and makes the correct prediction.

In summary, OCD achieves a Precision of 64.0% and a Recall of 17.1% on the Whole dataset and outperforms the two baselines by significant margins.

7.2 RQ2: The Effects of Main Components in OCD

Motivation: The key to JIT obsolete comment detection and update is to effectively capture the references and relationships between code changes and comments. To meet this need, in OCD and CUP, we adopt a co-attention mechanism, build a unified vocabulary, and use the word embeddings pre-trained by fastText for better representing, linking and fusing the information in code changes and comments. In this research question, we aim to understand the effects of the three components on OCD's performance. Besides, the encoder of OCD also adopts a special tokenizer for preserving comment format. We do not investigate its impact since we believe that it is indispensable for this task.
Approach: We compare OCD with its three variants, i.e., OCD-co-attn, OCD-uni-vocab and OCD-fastText. The performance differences between OCD and these variants can highlight the impacts of the three components.
Results: Table 5 shows the results of our comparisons. We can see that OCD outperforms all variants in terms of all metrics. Specifically, the improvements achieved by OCD over OCD-fastText are statistically significant and have large effect sizes in terms of all metrics, and OCD also significantly outperforms OCD-co-attn with at least medium effect sizes in terms of Precision and F1-Score. These results mean both the co-attention layer and the fastText pre-trained word embeddings are useful for obsolete comment detection. It can be seen that the performance improvement of OCD over OCD-co-attn in Recall is not significant. One possible reason for this is that sometimes the code-comment relationships captured by the co-attention layer may be too strict and explicit, resulting in the decrease of Recall. In addition, although the average performance of OCD is better than that of OCD-uni-vocab, our statistical analysis shows the performance differences between them are not significant. This is because we freeze the pre-trained word embeddings in OCD, which already ensures the same token in code and comments has the same embedding and makes the unified vocabulary somehow unnecessary for OCD. We still choose to use a unified vocabulary instead of two separate ones in OCD because a unified vocabulary can be helpful when the word embeddings are not frozen and we prefer to use the same encoder in OCD and CUP. Also, as we will see in Section 7.4, this component has positive effects on CUP's performance.

TABLE 6
Comparisons of CUP with three baselines

Approach	Accuracy	Recall@5	GLEU	AED	RED
Origin	0.0% (0)	N/A	49.3	3.59	1.000
NNUpdater	1.0% (98)	1.4%	10.7	17.69	8.544
FracoUpdater	1.7% (162)	N/A	50.4	3.62	1.019
CUP	**18.1% (1747)**	**26.1%**	**58.9**	**3.43**	**0.964**

*The numbers in brackets are the numbers of generated *correct comments*.

TABLE 7
P-Value and Cliff's Delta of CUP compared with the baselines

Approach	Accuracy	Recall@5	GLEU	AED	RED
Imp.Origin	****(L)	N/A	****(L)	****(L)	***(L)
Imp.NNUpdater	****(L)	****(L)	****(L)	****(L)	****(L)
Imp.FracoUpdater	****(L)	N/A	****(L)	****(L)	****(L)

In summary, the co-attention layer and the fastText pre-trained word embeddings can boost the effectiveness of OCD.

7.3 RQ3: The Effectiveness of CUP

Motivation: We want to investigate the effectiveness of CUP on JIT comment update.
Approach: We evaluate and compare CUP and its baselines on the Update dataset in terms of Accuracy, Recall@5, GLEU, AED, and RED.
Results: The evaluation results are shown in Table 6 and Table 7. We can see that CUP significantly outperforms all the baselines in terms of all evaluation metrics with large effect sizes. On average, it can correctly update comments in 1747 (18.1%) cases on the first tries, 10 times more than the best-performing baseline, and can generate *correct comments* for 26.1% of the test samples within 5 attempts.

Large improvements are achieved by CUP over NNUpdater in terms of all metrics. When compared to Origin and FracoUpdater, CUP performs much better on Accuracy, Recall@5 and GLEU, and also outperforms them in terms of AED and RED by substantial margins. These results indicate CUP can update comments more effectively and accurately than the three baselines. In addition, CUP is the only approach of which the AED is less than Origin's AED and the RED is less than 1. This highlights that CUP can be expected to help developers better understand where and how to update comments and reduce their edits on JIT comment updates.

To further figure out the reasons for CUP's better performance, we manually inspect the test results. Based on our inspection, we find that compared to NNUpdater and FracoUpdater, CUP has two major advantages:

First, CUP can learn and apply diverse comment update patterns automatically, while NNUpdater and FracoUpdater are limited to specific types of comment updates.

TABLE 8
Updater test sample 1

Code Change:

```
- public Document getUsersInRole(String role) throws
    ServletException, IOException {
+ public List<String> getUsersInRole(String role) {
  IUserRoleListService service = PentahoSystem.get(
      IUserRoleListService.class);
-   Element rootElement = new DefaultElement("users");
-   Document doc = DocumentHelper.createDocument(
    rootElement);
-   if (service != null) {
-     List<String> users = service.getUsersInRole(null,
    role);
-     for (Iterator<String> usersIterator = users.
    iterator(); usersIterator.hasNext();) {
-       String username = usersIterator.next().toString()
    ;
-       if ((null != username) && (username.length() > 0)
    ) {
-         rootElement.addElement("user").setText(username
    );
-     } } }
-   return doc;
+   return service.getUsersInRole( null, role );
  }
```

Old Comment: Returns XML for list of Users for a given Role.
New Comment: Returns a list of Users for a given Role.
NNUpdater: Finds the role associated to the given name.
FracoUpdater: Returns XML for list of Users for a given Role.
CUP-co-attn: Returns XML for list of Users for a given Role.
CUP-uni-vocab: Returns XML for list of Users for a given Role.
CUP-fastText: Returns XML for list of Users for a given list.
CUP-pointer: Returns a list of Users for a given Role.
CUP: Returns a list of Users for a given Role.

Specifically, NNUpdater relies on repeating new comments between test and training samples to generate *correct comments*. It may work well on some specific cases but lacks the generalization ability. FracoUpdater is based on manually summarized rules. It can obtain accurate results on identifier-renaming-related comment updates, but cannot handle other types of updates, e.g., updates related to type changes. In contrast, CUP leverages a probabilistic model to learn common patterns of JIT comment updates from extant code-comment co-changes. The patterns learned by CUP are more diverse than those of NNUpdater and FracoUpdater, and can cover more samples. For example, Table 8 presents a test sample in the Update dataset. In this sample, the developer changed the behavior of this method to directly return a list of Users instead of constructing and returning an XML document. The old comment of this sample should be updated accordingly to reflect the change of the return value. Considering the lack of explicit reference to "Document" in the old comment, it is non-trivial to design and implement rules for this kind of cases. Both NNUpdater and FracoUpdater fail to perform the correct update, while CUP succeeds.

Second, CUP can update semantic references between code and comments. NNUpdater does not take code-comment relationships into consideration. FracoUpdater can detect some semantic matching between renamed identifiers and comment phrases, but its quick-fix rules cannot correctly update such matching. Different from them, CUP explicitly adopts some components, such as the co-attention mechanism and the unified vocabulary, to enable our seq2seq model to effectively capture the relationships between code and comments. Based on our manual inspection, CUP can update not only lexical but also some semantic code-comment references with code changes. Table 9 presents an example. We can see that the developer used "executeCosmosAction" instead of "retryOperationWithThrottling" to wrap and execute the "readDocument" function. Therefore, there was no retry and the corresponding description in the old comment should be removed. NNUpdater and FracoUpdater fail to handle this case, but CUP accurately identifies such description and removes it when generating the new comment.

To the best of our knowledge, CUP is the first approach dedicated to JIT obsolete comment update using carefully-crafted neural network models. We hope it can inspire and facilitate follow-up works in this direction.

TABLE 9
Updater test sample 2

Code Change:

```
- public ResourceResponse<Document> readMetadata(BlobId
    blobId) throws DocumentClientException {
+ ResourceResponse<Document> readMetadata(BlobId blobId)
    throws DocumentClientException {
  String docLink = getDocumentLink(blobId.getID());
  RequestOptions options = getRequestOptions(blobId.
      getPartition().toPathString());
-   return retryOperationWithThrottling(() ->
    documentClient.readDocument(docLink, options),
+   return executeCosmosAction(() -> asyncDocumentClient.
    readDocument(docLink, options).toBlocking().single()
    ,
        azureMetrics.documentReadTime);
  }
```

Old Comment: Read the blob metadata document in the CosmosDB collection, retrying as necessary.
New Comment: Read the blob metadata document in the CosmosDB collection.
NNUpdater: Deletes the blob from the specific bucket
FracoUpdater: Read the blob metadata document in the CosmosDB collection, retrying as necessary.
CUP-co-attn: Read the blob metadata document in the CosmosDB collection, retrying as necessary.
CUP-uni-vocab: Read the blob metadata document in the CosmosDB collection, retrying as necessary.
CUP-fastText: Read the blob metadata document in the async collection, retrying as necessary.
CUP-pointer: Read the blob metadata document in the TachyonDB collection, attempting as necessary.
CUP: Read the blob metadata document in the CosmosDB collection.

In summary, CUP significantly outperforms the three baselines with large effect sizes, and can be expected to help developers better understand where and how to update obsolete comments and reduce their edits on JIT comment updates.

7.4 RQ4: The Effects of Main Components in CUP

Motivation: Following RQ2, we also want to know how the co-attention layer, the unified vocabulary and the fastText pre-trained word embeddings affect CUP's performance. In addition, the decoder of CUP adopts pointer generators to handle OOV words, ease the generation of new comments

TABLE 10
Comparisons of CUP with four variants

Approach	Accuracy	Recall@5	GLEU	AED	RED
CUP-co-attn	14.3% (1384)	23.7%	56.6	3.55	1.024
CUP-uni-vocab	17.1% (1645)	25.2%	58.3	3.49	1.001
CUP-fastText	14.7% (1416)	22.6%	57.0	3.64	1.077
CUP-pointer	6.9% (662)	12.8%	53.3	4.32	1.381
CUP	**18.1% (1747)**	**26.1%**	**58.9**	**3.43**	**0.964**

*The numbers in brackets are the numbers of generated *correct comments*.

TABLE 11
P-Value and Cliff's Delta of CUP compared with the variants

Approach	Accuracy	Recall@5	GLEU	AED	RED
Imp.CUP-co-attn	****(L)	****(L)	****(L)	****(L)	****(L)
Imp.CUP-uni-vocab	**(L)	**(L)	***(L)	*(L)	*(L)
Imp.CUP-fastText	****(L)	****(L)	****(L)	****(L)	****(L)
Imp.CUP-pointer	****(L)	****(L)	****(L)	****(L)	****(L)

and preserve comment format. We also want to investigate their impacts.

Approach: Similar to RQ2, we compare CUP with its four variants, i.e., CUP-co-attn, CUP-uni-vocab, CUP-fastText and CUP-pointer.

Results: The results of our comparisons are presented in Table 10 and Table 11. We can see that CUP performs better than the four variants in terms of all metrics. The performance improvements achieved by CUP are all statistically significant with large effect sizes. For Accuracy, the improvements achieved by CUP range from 5.8% to 162%, and CUP can generate at least 102 more *correct comments* than the variants. These results indicate that the co-attention mechanism, the unified vocabulary, the fastText embeddings and the pointer generators are all useful and effective for JIT obsolete comment update. In addition, as we mentioned in Section 7.2, the frozen pre-trained word embeddings make the unified vocabulary less useful for OCD. However, the unified vocabulary plays an important role in CUP, because it makes the decoder of CUP able to represent and generate, instead of only copy, code-only tokens.

To better understand these performance differences, we manually inspect the test results of the four variants. We find that without the four components, CUP will capture more incorrect code-comment references, may more frequently miss updating comments, perform inaccurate updates on the right references, or fail to generate the tokens which are rare or can not be found in the training set but appear in the input. For example, in the samples presented in Table 8 and Table 9, both CUP-co-attn and CUP-uni-vocab fail to predict the proper updates and regard modifying nothing as the best solution. CUP-fastText builds incorrect references for the two samples. It builds a reference between "Role" and "List" in Sample 1, links "CosmosDB" with "async" in Sample 2, and performs incorrect comment updates. Although CUP-pointer correctly updates the comment for Sample 1, it fails to generate "CosmosDB" and performs an inaccurate update on the right reference (i.e., "retrying as necessary") in Sample 2. These results demonstrate that the four components all play important roles in capturing and updating code-comment references.

TABLE 12
Comparisons of CUP^{2}with the baseline

Appr	P	R	F1	#TPC/#TP	#FPC/#FP	AED	RED
Fraco	21.7%	14.6%	17.4%	162/1406	**1602**/5076	2.58 (0.61)	2.182
CUP^2	**64.0%**	**17.1%**	**27.0%**	**698**/1650	381/932	**1.86** (1.95)	**0.842**
Imp	****(L)	****(L)	****(L)	****(L)	***(L)	****(L)	****(L)

*__Appr__, __P__, __R__ and __F1__ refer to Approach, Precision, Recall and F1-Score, respectively.
*#TP and #FP refer to the numbers of true positive samples and false positive samples, respectively.
*The numbers in the brackets of the AED column are the AEDs between the old comments and the new comments.

In summary, the co-attention mechanism, the unified vocabulary, the fastText embeddings and the pointer generators are helpful for capturing and updating code-comment references and can improve the effectiveness of CUP.

7.5 RQ5: The Effectiveness of CUP^2

Motivation: We want to investigate how effective CUP^2 can be after combining the detection and the update of obsolete comments.

Approach: We evaluate and compare CUP^2 and its baseline, i.e., Fraco, on the Whole dataset described in Section 5.4 in terms of Detection Precision, Recall and F1-Score, #TPC/#TP, #FPC/#FP, AED and RED.

Results: Table 12 presents the evaluation results. Each number in the bracket is the AED between the old comments and the new comments of the samples detected by an approach. For convenience, we refer to such numbers as Origin AEDs. Because the sets of samples detected by FracoDetector and OCD are different, the Origin AEDs of Fraco and CUP^2 are also different.

We can see that CUP^2 outperforms Fraco in terms of all metrics but #FPC by substantial margins. The performance improvements of CUP^2 over Fraco in terms Precision, Recall, F1-Score, #TPC/#TP, #FPC/#FP, AED and RED are all statistically significant and have large effect sizes. At the detection stage, CUP^2 detects more obsolete comments with higher precision, as we discussed in RQ1. At the update stage, CUP^2 updates both TP samples and FP samples with higher accuracy than Fraco. In detail, the #TPC of CUP^2 is about 4.3 times of Fraco, and CUP^2 achieves an accuracy of 42.3% (698/1650) on its TP samples, much better than that accuracy of Fraco (11.5%). Fraco's #FPC is greater than that of CUP^2. However, the accuracy of CUP^2 on its FP samples is 41.0%, still better than that of Fraco (31.6%). Furthermore, the overall accuracy, i.e., (#TPC+#FPC)/(#TP+#FP), of CUP^2 (41.8%) is much higher than that of Fraco (27.2%). As for AED and RED, Fraco's AED is much higher than its Origin AED, which is only 0.61 since Fraco detects many false positive samples, and Fraco's RED is greater than 1. In contrast, the AED of CUP^2 is lower than its Origin AED, and its RED is less than 1.

We admit that the performance of CUP^2 is not perfect, but we argue that CUP^2 is still useful for developers and researchers, because: First, although the Recall of CUP^2's detection stage is 17.1%, its Precision is 64.0%, which is reasonable. We believe that for obsolete comment detection

TABLE 13
A false negative sample

Code Change:

```
@@ -1,5 +1,6 @@
- private MountPointInfo getMountPointInfo(MountInfo
    mountInfo) {
-    MountPointInfo info = mountInfo.toMountPointInfo();
+ private MountPointInfo getMountPointInfoInternal(
    MountInfo mountInfo, boolean useDisplayValues) {
+   MountPointInfo info = useDisplayValues
+       ? mountInfo.toDisplayMountPointInfo() : mountInfo
    .toMountPointInfo();
    try (CloseableResource<UnderFileSystem> ufsResource =
          mUfsManager.get(mountInfo.getMountId()).
        acquireUfsResource()) {
      UnderFileSystem ufs = ufsResource.get();
    ...
```

Old Comment: Gets the mount point information from a mount information.

New Comment: Gets the mount point information for display from a mount information.

TABLE 14
A false positive sample

Code Change:

```
- public static String join(Collection collection, String
    separator) {
-   return collection == null ? null : join(collection.
    iterator(), separator);
+ public static String join(Iterable iterable, String
    separator) {
+   return iterable == null ? null : join(iterable.
    iterator(), separator);
  }
```

Old Comment: Joins the elements of the provided into a single String containing the provided elements.

New Comment: Joins the elements of the provided into a single String containing the provided elements.

2CUP: Joins the elements of the provided into a single String containing the provided iterable.

TABLE 15
A failed true positive sample

Code Change:

```
- protected UIInputListener createUIInputListener ()
+ protected ButtonInputListener<C>
    createButtonInputListener ()
  {
-   return new WButtonListener ();
+   return new WButtonInputListener<C, WButtonUI<C>> ();
  }
```

Old Comment: Returns {@link UIInputListener} for the button.

New Comment: Returns {@link ButtonInputListener} for the {@link AbstractButton}.

2CUP: Returns {@link ButtonInputListener} for the button.

and update, Precision is more important than Recall. Since even if an approach only predicts one comment, if the result is correct, this approach can be helpful for developers. In contrast, if an approach achieves 100% Recall but to find one obsolete comment developers need to inspect 10 candidates, developers may not be happy to use it. Second, we argue that the overall accuracy of 41.8% with RED of 0.842 indicates that CUP^2 can help developers to quickly understand the specific locations and operations of comment updates and reduce their edits on updating comments. Third, to the best of our knowledge, CUP^2 is the first work focusing on JIT obsolete comment detection and update. The task itself is not easy considering the difficulties of "understanding" and "linking" code changes and comments, and CUP^2 has outperformed the baseline in all metrics by significant margins. Based on these facts, we believe CUP^2 can promote the development of this research direction and inspire other researchers to tackle this important task.

In summary, CUP^2 outperforms the rule-based baseline in terms of almost all metrics by significant margins. It can be useful for developers in identifying obsolete comments, understanding the specific locations and operations of obsolete comment updates and reducing their edits on JIT comment updates.

8 Discussion

8.1 When Does CUP^2 Fail

The test set of the Whole dataset contains 454K samples. According to our evaluation results presented in Section 7.5, at the detection stage, CUP^2 produces 1,650 true positive (TP) samples, 932 FP samples, 7,997 false negative (FN) samples and 443K true negative (TN) samples. All TN samples are correctly handled by CUP^2. However, for the other three kinds of samples, the two-stage procedure of CUP^2 results in three types of errors that hinder it from performing perfect obsolete comment repair.

First, CUP^2 will not perform updates for the 7997 FN samples. Table 13 presents an example. In this sample, a flag was added to this method's parameter list to control the type of the returned information, i.e., either display info or raw info. The comment was updated to explicitly mention "display". OCD predicts that the old comment does not need to be updated. Although the prediction is wrong, we think it is excusable because there is no obvious inconsistency between the new code and the old comment, and the comment update performed by developers can somehow be considered as an optimization of language and expression.

Second, for FP samples, the comments generated at the update stage of CUP^2 may be different from the corresponding old comments. This type includes 551 samples, one of which is presented in Table 14. In this sample, the developer changed the first parameter of this method, replacing the "Collection" with an "Iterable". The comment was still consistent with the code, hence it did not require updates. However, CUP^2 predicts this sample to be positive and updates the old comment by replacing the "elements" with "iterable". The possible reason for these incorrect behaviors is that CUP^2 links the "elements" in the comment with the "collection" in the code, which is correct, but fails to capture the semantic equivalence of "elements" and "iterable" in this context.

Third, at the update stage, CUP^2 may fail to correctly update TP samples, namely failed TP samples. 952 test samples belong to this type. Table 15 presents an example. In this example, the developer changed the behavior of this method to create a "ButtonInputListener" instead of a "UIInputListener", and made the corresponding update on

TABLE 16
The effectiveness of the three ways to tackle class imbalance

Approach	Precision	Recall	F1-Score
$OCD_{balanced}$	11.4%	67.3%	19.5%
OCD_{sample}	10.5%	**68.5%**	18.2%
$OCD_{focal\text{-}loss}$	**71.6%**	5.7%	10.5%
OCD	64.0%	17.1%	**27.0%**

the comment. In addition, the developer also optimized the old comment by replacing "button" with "AbstractButton", which is the base class of "C". CUP successfully replaces the "UIInputListener" in the comment with "ButtonInputListener". However, because "AbstractButton" cannot be inferred from the method change and the old comment, CUP fails to update the "button" in the comment.

Section 7.5 has shown that CUP^2 is effective in JIT obsolete comment detection and update. We believe devising more advanced obsolete comment detectors and updaters to reduce the aforementioned errors is an interesting and promising direction for future work.

8.2 Class Imbalance of the Whole Dataset

As we mentioned in Section 5.4, the Whole dataset is highly class imbalanced (the ratio of the positive samples to the negative samples is about 1/38). Therefore, we also explore some commonly-used ways to alleviate the class imbalance problem and construct several variants for OCD:

$OCD_{balanced}$: We build a balanced training set by selecting all positive samples and sampling the same number of negative samples from the training set of the Whole dataset. The architecture of $OCD_{balanced}$ is the same as OCD, but it is trained on this balanced training set.

OCD_{sample}: While the architecture of OCD_{sample} is also identical to OCD, when training, OCD_{sample} constructs each batch by sampling half a batch of positive samples and selecting half a batch of negative samples from the training set of the Whole dataset. It can be seen that for OCD_{sample}, the number of available negative samples during training is much larger than that for $OCD_{balanced}$.

$OCD_{focal\text{-}loss}$: The only difference between $OCD_{focal\text{-}loss}$ and OCD is that $OCD_{focal\text{-}loss}$ uses focal loss [65] instead of cross entropy as the loss function. Focal loss is originally designed to tackle the severe class imbalance in the image object detection task, and is shown to be effective for many classification tasks [66], [67], [68]. Its core idea is to pay more attention to hard samples and less attention to easy samples during training. It is computed as follows:

$$FL(p_t) = -\alpha_t(1-p_t)^{\gamma}\log(p_t)$$

For each sample, p_t and α_t are respectively the probability estimated by a model and the weighting factor for its ground truth class, and γ is the focusing parameter which controls the strength of the modulation. The original paper of focal loss [65] showed that setting γ to 2.0 and α_t to 0.25 and 0.75 for the positive and the negative classes makes focal loss work best. We follow [65] and use the same parameters. $OCD_{focal\text{-}loss}$ is trained on the Whole dataset.

All three variants are validated and evaluated on the Whole dataset. Table 16 presents the evaluation results. It can be seen that the Recalls of $OCD_{balanced}$ and OCD_{sample} are much better than that of OCD, but their Precisions are much worse and OCD also outperforms them in terms of F1-score by substantial margins. Although $OCD_{focal\text{-}loss}$ performs better than OCD in terms of Precision, its Recall and F1-Score are only 5.7% and 10.5%, far lower than those of OCD. These results indicate that the three common ways to alleviate the class imbalance problem, i.e., training on a balanced dataset, sampling balanced batches during training and using focal loss, can not improve OCD. One possible explanation is that the primary obstacle of the obsolete comment detection task is the inherent difficulty of "understanding" code changes and comments. Based on these results, CUP^2 uses OCD, instead of the three variants, for detection. That being said, the class imbalance problem is still an important challenge for obsolete comment detection and it calls for investigation and design of more advanced techniques in future studies.

TABLE 17
The effectiveness of the synergy between OCD and CUP

Approach	Accuracy	Recall@5	GLEU	AED	RED
+OCD.vocab	17.6% (1697)	**26.1%**	58.6	**3.40**	**0.954**
+OCD.encoder	15.9% (1531)	24.1%	57.6	3.51	1.010
CUP	**18.1% (1747)**	**26.1%**	**58.9**	3.43	0.964

*+OCD.vocab and +OCD.encoder refer to CUP+OCD.vocab and CUP+OCD.encoder, respectively.

8.3 Synergy Between OCD and CUP

Considering that OCD and CUP share the same encoder architecture, we also investigate the potential synergy between OCD and CUP. Specifically, we construct two variants CUP+OCD.vocab and CUP+OCD.encoder for CUP and one variant $CUP^2_{combined}$ for CUP^2:

CUP+OCD.vocab uses the same architecture as CUP but with the unified vocabulary built from the Whole dataset instead of the Update dataset.

CUP+OCD.encoder reuses the encoder learned by OCD before training, also uses the vocabulary built from the Whole dataset, and keeps other parts identical to CUP.

$CUP^2_{combined}$ is a single model that contains only one encoder but both the output components of OCD and CUP. Given a batch of code-comment change samples, when training, $CUP^2_{combined}$ first encodes each sample into feature vectors, and then predicts the probability that the old comment should be updated and generates a new comment for it. The training loss of a batch is the average of all samples' classification losses and the generation losses of the samples whose label is positive. When inferring, after encoding each sample, $CUP^2_{combined}$ first detects positive samples using OCD's output component and only generates new comments for those samples predicted to be positive.

CUP's variants are trained and tested on the Update dataset. Experimental results are shown in Table 17. We can see that the performance of CUP, CUP+OCD.vocab and CUP+OCD.encoder is close. CUP+OCD.vocab slightly outperforms CUP in terms of AED and RED, behind which the rationale is that OCD's large vocabulary reduces the number of OOV words and eases the generation of some rare words. However, CUP achieves better Accuracy and GLEU than and equal Recall to CUP+OCD.vocab, and

TABLE 18
The effectiveness of the combined model $CUP^2_{combined}$

Appr	P	R	F1	#TPC/#TP	#FPC/#FP	AED	RED
Comb	55.8%	**18.9%**	**28.3%**	363/1826	**829**/1449	2.17 (1.73)	1.011
CUP²	**64.0%**	17.1%	27.0%	**698**/1650	381/932	**1.86** (1.95)	**0.842**

*Comb refers to $CUP^2_{combined}$.

performs better than CUP+OCD.encoder in all metrics. In addition, on our server, it takes 120 mins, 189 mins and 168 mins, respectively, to train CUP, CUP+OCD.vocab and CUP+OCD.encoder once, which means transferring knowledge from OCD to CUP increases CUP's training time substantially. One possible reason for the decreases in performance and efficiency is that OCD and CUP have related but different focuses. For example, OCD may care more about the differences between the overall similarity of <old comment, old code> and that of <old comment, new code>, while CUP may focus on specific references between the old comment and the code change in order to perform accurate comment updates. If we transfer the knowledge learned by OCD to CUP, CUP may be confused and it may take more time for CUP to correct such knowledge than to learn from scratch. To sum up, using the vocabulary and transferring the encoder of OCD do not improve the effectiveness or the efficiency of CUP.

$CUP^2_{combined}$ is trained and tested on the Whole dataset. Table 18 presents the evaluation results. We can see that compared to CUP^2, for obsolete comment detection, $CUP^2_{combined}$ obtains a slightly better Recall and F1-score, but worse Precision. As we discussed in Section 7.5, we argue Precision is more important than Recall for this task. For comment update, the #FPC and the #FPC/#FP of $CUP^2_{combined}$ are better than those of CUP^2, which means $CUP^2_{combined}$ is more accurate on FP samples. However, CUP^2 outperforms $CUP^2_{combined}$ in terms of all other metrics, i.e, #TPC, #TPC/#TP, AED and RED. Specifically, CUP^2's #TPC (698) and #TPC/#TP (42.3%) are over 1.9 times of those of CUP^2 (363 and 19.9%), respectively, and its RED is less than 1 while $CUP^2_{combined}$'s RED is not. Moreover, it takes about 18 hours to train CUP^2 but 78 hours to train $CUP^2_{combined}$. We think the possible rationales behind these phenomenons are: First, as we mentioned above, OCD and CUP have related but different focuses. Therefore, combining OCD and CUP in a single model may confuse each other, resulting in no significant performance improvement and a heavy increase in training time. Second, the model size of $CUP^2_{combined}$ is larger than CUP and we can only train it with the batch size of at most 28 under the limitation of our GPU memory. With the limited batch size and the highly imbalanced data, $CUP^2_{combined}$ may sometimes be trained on a batch with only negative samples, which may hinder the learning process. Based on these results, we do not combine OCD and CUP as a single model.

In summary, our attempts, i.e., CUP+OCD.vocab, CUP+OCD.encoder and $CUP^2_{combined}$, fail to bring either performance or efficiency improvements. However, this does not mean the synergy between the detection and the update of obsolete comments and advanced multi-task models are not helpful for this task. We believe further investigations of such synergy and advanced multi-task models/frameworks for this task can be interesting and promising directions for future work.

8.4 Threats to Validity

First, our dataset is built only from Java projects and focuses on method comments, which may not be representative of all programming languages and comment types. However, Java is one of the most popular programming languages. Method comments are an important type of comments and are often referred to by developers for program comprehension. Besides, OCD and CUP are independent of programming languages and comment types. After being trained on proper datasets, they can be applied to process code changes of other programming languages and generate other types of comments.

The second threat is related to the method mappings we build from commits. Before extracting modified methods from a commit, we use GumTree to match the methods in the two revisions. However, some method mappings identified by GumTree are suboptimal. We mitigate this threat by 1) adding a new phase in GumTree to improve its accuracy in method matching, 2) filtering the samples with abstract methods, which are often mismatched. Also, we manually checked 200 samples in the test set and only found one suboptimal method mapping. Therefore, we believe this threat is limited.

Another threat is about the hyper-parameter choices for RandomForest. To reliably re-implement Liu et al.'s approach [16] and fairly compare OCD with it, we keep the settings and hyper-parameters of RandomForest the same as those used by Liu et al. Such settings and hyper-parameters might not be the best for this work. However, firstly, this work targets at a very similar task to Liu et al.'s work. Secondly, all the settings and hyper-parameters used in Liu et al.'s work is in line with the default ones recommended by the widely-used machine learning library scikit-learn [58]. Thus, we believe the hyper-parameters choices of RandomForest are reasonable.

Besides, due to the existence of obsolete comments in software repositories, the Whole dataset may contain a few mislabeled samples, i.e., the samples where the old comments should be updated with the code changes but were not. Prior studies [27] have shown that over 97% of comment changes are performed with the corresponding code changes. We also manually checked 200 samples that were randomly selected from the test set of the Whole dataset and did not find any mislabeled sample. Therefore, we believe that this threat is minimal. It is worth mentioning that the limited number of noisy samples in our dataset does not mean CUP^2 is not useful. As shown by prior work, the introduced obsolete comments can result in bugs in the future and are harmful to the robustness of a software system [8], [9], [10], [11], [12]. In addition, CUP^2 can be integrated into IDEs and provide recommendations just after developers edit code. Even if developers know that they should update comments before committing code changes, it takes time for them to identify the comments requiring update, the sentences and the specific phrases that should be updated, and figure out how to update. By predicting obsolete comments and recommending specific update operations, CUP^2 and other obsolete comment detection and

TABLE 19
A valid comment generated by CUP

Code Change:

```
- public RecordingCatchUpSupplier
     recordingCatchUpSupplier()
+ public LogCatchUpSupplier logCatchUpSupplier()
  {
-    return recordingCatchUpSupplier;
+    return logCatchUpSupplier;
  }
```

Old Comment: The {@link RecordingCatchUpSupplier} to use for catching up recordings.

New Comment: The {@link LogCatchUpSupplier} to use for catching up log recordings.

CUP: The {{@link LogCatchUpSupplier} to use for catching up recordings.

TABLE 20
A valid comment generated by CUP^2

Code Change:

```
- public ZonedDateTime plus(TemporalAdder adder) {
-    return (ZonedDateTime) adder.addTo(this);
+ public ZonedDateTime plus(TemporalAmount amount) {
+    return (ZonedDateTime) amount.addTo(this);
  }
```

Old Comment: Returns a copy of this date-time with the specified period added.

New Comment: Returns a copy of this date-time with the specified period added.

CUP^2: Returns a copy of this date-time with the specified amount added.

update approaches can help developers quickly identify obsolete comments and understand where and how to perform comment updates. Considering that CUP^2 achieves a RED of 0.842 and applying recommended comment updates only requires a single click by developers, we believe CUP^2 can also reduce developers' edits on obsolete comment updates. In addition, a recent study [69] also confirmed that it is hard for developers to manually identify obsolete comments hidden in a large code base and developers appreciate the tools and efforts to detect and fix obsolete comments.

It is also worth noting that the Accuracy metric for CUP and CUP^2 is strict, because it only rewards the generated comments identical to the ground truth. Since there may exist some generated comments that are different from the ground truth but also valid, the Accuracy metric in fact measures the performance lower bound instead of the real-world accuracy. Unfortunately, there is no automatic way to evaluate the real-world accuracy of comment updaters. To mitigate this threat, first, we also use the GLEU metric, which is widely used to evaluate Grammatical Error Correction (GEC) systems and is more flexible than Accuracy, to evaluate CUP. According to Section 7.3, CUP also outperforms all the baselines in terms of GLEU. Then, we manually inspect the results of CUP, NNUpdater and FracoUpdater on 100 randomly selected test samples and the update results of CUP^2 and Fraco on 100 test samples that are predicted to be positive by their own detectors and are randomly selected. Specifically, given a selected test sample, we carefully read its code change, old comment and new comment. If the comment generated by an approach is not the same as the new comment (i.e., not correct), we read it and compare it with the new comment to decide whether it is valid (i.e., semantically identical to the new comment). We find that CUP, NNUpdater and FracoUpdate generate 3, 1 and 0 valid comments, respectively. Table 19 presents a valid comment generated by CUP. We can see that CUP successfully replaces "RecordingCatchUpSupplier" with "LogCatchUpSupplier", but fails to insert "log". However, since one can know the "recordings" are "log recordings" according to "LogCatchUpSupplier", this generated comment is semantically identical to the new comment, i.e., is a valid comment. For CUP^2 and Fraco, the numbers of generated valid comments are 10 and 8. Table 20 shows a valid comment generated by CUP^2. In this sample, developers did not update the comment with the code change. But OCD predicts this comment to be obsolete and CUP replaces "period" in the comment with "amount". Since "period" and "amount" are synonymous in the context of "ZonedDateTime", the comment generated by CUP^2 is a valid comment. Those results indicate that valid comments do exist and can make the real-world accuracy of CUP and CUP^2 better than their Accuracy values, but we believe such valid comments do not affect the conclusions of our evaluation, i.e., CUP and CUP^2 significantly outperforms their own baselines.

9 Conclusion and Future Work

To reduce and avoid obsolete comments in source code, this work proposes to detect and update obsolete comments in time with code changes. We propose a two-stage framework named CUP^2 to automate this task. CUP^2 takes as input code changes and their associated comments, leverages a detector named OCD to predict whether each comment requires updates, and then updates predicted obsolete comments using a comment updater named CUP. OCD (CUP) adopts a distinct neural network model to learn the features of obsolete comments (the patterns of comment updates) from massive code-comment change samples and perform obsolete comment prediction (update) automatically. Comprehensive experiments on a large dataset with over 4 million code-comment change samples show that: 1) OCD, CUP and CUP^2 outperform their own baselines by significant margins, and 2) CUP^2 can help developers detect obsolete comments, understand where and how to update obsolete comments and reduce their edits on comment updates.

In the future, we plan to conduct an in-field study to investigate the usefulness of CUP^2 within an industrial setting. We also plan to adapt CUP^2 to other code granularity, such as statements, and other comment types, e.g., inner comments of methods, with more context, such as the fields and the other methods of the corresponding classes, considered. In addition, as the first attempt on the JIT obsolete comment detection and update task, CUP^2, including both OCD and CUP, is not perfect and there is still much space for improvements. On the one hand, due to the limitation of LSTM, CUP^2 may not be able to well handle the samples of which the code changes are long and complicated. Thus, it would be interesting and promising to propose more advanced obsolete comment detectors and updaters to improve CUP^2's effectiveness further. On the other hand, designing advanced multi-task learning methods to enhance

the synergy between OCD and CUP is also a potential way towards better performance.

Replication Package: https://github.com/Tbabm/CUP2

REFERENCES

[1] Y. Padioleau, L. Tan, and Y. Zhou, "Listening to programmers taxonomies and characteristics of comments in operating system code," in *Proceedings of the 31st International Conference on Software Engineering*, 2009, pp. 331–341.

[2] D. Steidl, B. Hummel, and E. Juergens, "Quality analysis of source code comments," in *Proceedings of the 21st International Conference on Program Comprehension*, 2013, pp. 83–92.

[3] L. Pascarella, M. Bruntink, and A. Bacchelli, "Classifying code comments in java software systems," *Empirical Software Engineering*, pp. 1–39, 2019.

[4] S. N. Woodfield, H. E. Dunsmore, and V. Y. Shen, "The effect of modularization and comments on program comprehension," in *Proceedings of the 5th International Conference on Software Engineering*, 1981, pp. 215–223.

[5] A. T. Ying, J. L. Wright, and S. Abrams, "Source code that talks: an exploration of eclipse task comments and their implication to repository mining," in *Proceedings of the International Workshop on Mining Software Repositories*, 2005, pp. 1–5.

[6] S. C. B. de Souza, N. Anquetil, and K. M. de Oliveira, "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*, 2005, pp. 68–75.

[7] "A commit in apache kafka," https://github.com/apache/kafka/commit/9dc76f8872b862ca008562cdcf8cf50524e2eaa3, 2020.

[8] L. Tan, D. Yuan, and Y. Zhou, "Hotcomments: how to make program comments more useful?" in *Proceedings of the 11th USENIX Workshop on Hot Topics in Operating Systems*, 2007, pp. 1–6.

[9] L. Tan, D. Yuan, G. Krishna, and Y. Zhou, "/* icomment: Bugs or bad comments?*," in *Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles*, 2007, pp. 145–158.

[10] D. L. Parnas, "Precise documentation: The key to better software," in *The Future of Software Engineering*. Springer, 2011, pp. 125–148.

[11] S. H. Tan, D. Marinov, L. Tan, and G. T. Leavens, "@ tcomment: Testing javadoc comments to detect comment-code inconsistencies," in *Proceedings of the 5th International Conference on Software Testing, Verification and Validation*, 2012, pp. 260–269.

[12] W. M. Ibrahim, N. Bettenburg, B. Adams, and A. E. Hassan, "On the relationship between comment update practices and software bugs," *Journal of Systems and Software*, vol. 85, no. 10, pp. 2293–2304, 2012.

[13] L. Tan, Y. Zhou, and Y. Padioleau, "acomment: mining annotations from comments and code to detect interrupt related concurrency bugs," in *Proceedings of the 33rd International Conference on Software Engineering*, 2011, pp. 11–20.

[14] G. Sridhara, "Automatically detecting the up-to-date status of todo comments in java programs," in *Proceedings of the 9th India Software Engineering Conference*, 2016, pp. 16–25.

[15] I. K. Ratol and M. P. Robillard, "Detecting fragile comments," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, 2017, pp. 112–122.

[16] Z. Liu, H. Chen, X. Chen, X. Luo, and F. Zhou, "Automatic detection of outdated comments during code changes," in *Proceedings of the 42nd Annual Computer Software and Applications Conference*, vol. 1, 2018, pp. 154–163.

[17] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation," in *Proceedings of the 26th International Conference on Program Comprehension*, 2018, pp. 200–210.

[18] M. Tufano, J. Pantiuchina, C. Watson, G. Bavota, and D. Poshyvanyk, "On learning meaningful code changes via neural machine translation," in *Proceedings of the 41st International Conference on Software Engineering*, 2019, pp. 25–36.

[19] Z. Liu, X. Xia, C. Treude, D. Lo, and S. Li, "Automatic generation of pull request descriptions," in *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*, 2019, pp. 176–188.

[20] C. Napoles, K. Sakaguchi, M. Post, and J. Tetreault, "Gleu without tuning," *arXiv preprint arXiv:1605.02592*, 2016.

[21] Z. Liu, X. Xia, Y. Meng, and S. Li, "Automating just-in-time comment updating," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 2020, pp. 713–725.

[22] C. Chen, Z. Xing, and Y. Liu, "By the community & for the community: a deep learning approach to assist collaborative editing in q&a sites," *Proceedings of the ACM on Human-Computer Interaction*, vol. 1, no. CSCW, pp. 1–21, 2017.

[23] T. Ge, F. Wei, and M. Zhou, "Fluency boost learning and inference for neural grammatical error correction," in *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2018, pp. 1055–1065.

[24] S. Chollampatt and H. T. Ng, "A multilayer convolutional encoder-decoder neural network for grammatical error correction," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 32, no. 1, 2018.

[25] C. Napoles, K. Sakaguchi, M. Post, and J. Tetreault, "Ground truth for grammatical error correction metrics," in *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 2: Short Papers)*, 2015, pp. 588–593.

[26] Z. M. Jiang and A. E. Hassan, "Examining the evolution of code comments in postgresql," in *Proceedings of the International Workshop on Mining Software Repositories*, 2006, pp. 179–180.

[27] B. Fluri, M. Wursch, and H. C. Gall, "Do code and comments co-evolve? on the relation between source code and comment changes," in *Proceedings of the 14th Working Conference on Reverse Engineering*, 2007, pp. 70–79.

[28] B. Fluri, M. Würsch, E. Giger, and H. C. Gall, "Analyzing the co-evolution of comments and source code," *Software Quality Journal*, vol. 17, no. 4, pp. 367–394, 2009.

[29] M. Linares-Vásquez, B. Li, C. Vendome, and D. Poshyvanyk, "How do developers document database usages in source code?" in *Proceedings of the 30th International Conference on Automated Software Engineering*, 2015, pp. 36–41.

[30] F. Wen, C. Nagy, G. Bavota, and M. Lanza, "A large-scale empirical study on code-comment inconsistencies," in *Proceedings of the 27th International Conference on Program Comprehension*, 2019, pp. 53–64.

[31] Y. Zhou, R. Gu, T. Chen, Z. Huang, S. Panichella, and H. Gall, "Analyzing apis documentation and code to detect directive defects," in *Proceedings of the 39th International Conference on Software Engineering*, 2017, pp. 27–37.

[32] H. Malik, I. Chowdhury, H.-M. Tsou, Z. M. Jiang, and A. E. Hassan, "Understanding the rationale for updating a function's comment," in *Proceedings of the International Conference on Software Maintenance*, 2008, pp. 167–176.

[33] S. Panthaplackel, P. Nie, M. Gligoric, J. J. Li, and R. J. Mooney, "Learning to update natural language comments based on code changes," *arXiv preprint arXiv:2004.12169*, 2020.

[34] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker, "Towards automatically generating summary comments for java methods," in *Proceedings of the International Conference on Automated Software Engineering*, 2010, pp. 43–52.

[35] L. Moreno, J. Aponte, G. Sridhara, A. Marcus, L. Pollock, and K. Vijay-Shanker, "Automatic generation of natural language summaries for java classes," in *Proceedings of the 21st International Conference on Program Comprehension*, 2013, pp. 23–32.

[36] S. Haiduc, J. Aponte, L. Moreno, and A. Marcus, "On the use of automated text summarization techniques for summarizing source code," in *Proceedings of the 17th Working Conference on Reverse Engineering*, 2010, pp. 35–44.

[37] E. Wong, T. Liu, and L. Tan, "Clocom: Mining existing source code for automatic comment generation," in *Proceedings of the 22nd International Conference on Software Analysis, Evolution, and Reengineering*, 2015, pp. 380–389.

[38] N. Nazar, Y. Hu, and H. Jiang, "Summarizing software artifacts: A literature review," *Journal of Computer Science and Technology*, vol. 31, no. 5, pp. 883–909, 2016.

[39] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, "Summarizing source code using a neural attention model," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics*, 2016, pp. 2073–2083.

[40] Y. Wan, Z. Zhao, M. Yang, G. Xu, H. Ying, J. Wu, and P. S. Yu, "Improving automatic source code summarization via deep reinforcement learning," in *Proceedings of the 33rd International Conference on Automated Software Engineering*, 2018, pp. 397–407.

[41] A. LeClair, S. Jiang, and C. McMillan, "A neural model for generating natural language summaries of program subroutines," in *Proceedings of the 41st International Conference on Software Engineering*, 2019, pp. 795–806.

[42] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation with hybrid lexical and syntactical information," *Empirical Software Engineering*, pp. 1–39, 2019.

[43] Y. Zhou, X. Yan, W. Yang, T. Chen, and Z. Huang, "Augmenting java method comments generation with context information based on neural networks," *Journal of Systems and Software*, vol. 156, pp. 328–340, 2019.

[44] C. Qiuyuan, X. Xin, H. Han, L. David, and L. Shanping, "Why my code summarization model does not work: code comment improvement with category prediction," *ACM Transactions on Software Engineering and Methodology*, 2020.

[45] R.-M. Karampatsis, H. Babii, R. Robbes, C. Sutton, and A. Janes, "Big code!= big vocabulary: Open-vocabulary models for source code," in *Proceedings of the 42nd IEEE/ACM International Conference on Software Engineering*, 2020, pp. 1073–1085.

[46] P. Yin, G. Neubig, M. Allamanis, M. Brockschmidt, and A. L. Gaunt, "Learning to represent edits," in *Proceedings of the 7th International Conference on Learning Representations*, 2018.

[47] E. Grave, P. Bojanowski, P. Gupta, A. Joulin, and T. Mikolov, "Learning word vectors for 157 languages," in *Proceedings of the International Conference on Language Resources and Evaluation*, 2018.

[48] M.-T. Luong, H. Pham, and C. D. Manning, "Effective approaches to attention-based neural machine translation," in *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, 2015, pp. 1412–1421.

[49] A. See, P. J. Liu, and C. D. Manning, "Get to the point: Summarization with pointer-generator networks," in *Proceedings of the Annual Meeting of the Association for Computational Linguistics*, 2017, pp. 1073–1083.

[50] D. Kramer, "API documentation from source code comments: A case study of Javadoc," in *Proceedings of the 17th Annual International Conference on Computer Documentation*, 1999, pp. 147–153.

[51] "Javaparser," https://javaparser.org/, 2020.

[52] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and accurate source code differencing," in *Proceedings of the 29th International Conference on Automated Software Engineering*, 2014, pp. 313–324.

[53] "Natural language toolkit nltk 3.5 documentation," http://www.nltk.org/, 2020.

[54] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals," in *Soviet Physics Doklady*, vol. 10, no. 8, 1966, pp. 707–710.

[55] M. Kim and D. Notkin, "Discovering and representing systematic code changes," in *Proceedings of the 31st International Conference on Software Engineering*, 2009, pp. 309–319.

[56] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. Al-Kofahi, and T. N. Nguyen, "Recurring bug fixes in object-oriented programs," in *Proceedings of the 32nd International Conference on Software Engineering*, 2010, pp. 315–324.

[57] N. Tsantalis, M. Mansouri, L. M. Eshkevari, D. Mazinanian, and D. Dig, "Accurate and efficient refactoring detection in commit history," in *Proceedings of the 40th International Conference on Software Engineering*, 2018, pp. 483–494.

[58] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg *et al.*, "Scikit-learn: Machine learning in python," *the Journal of machine Learning research*, vol. 12, pp. 2825–2830, 2011.

[59] Z. Liu, X. Xia, A. E. Hassan, D. Lo, Z. Xing, and X. Wang, "Neural-machine-translation-based commit message generation: how far are we?" in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 373–384.

[60] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: a method for automatic evaluation of machine translation," in *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, 2002, pp. 311–318.

[61] "Word vectors for 157 languages," https://fasttext.cc/docs/en/crawl-vectors.html, 2020.

[62] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *Proceedings of the 3rd International Conference on Learning Representations*, 2015.

[63] F. Wilcoxon, "Individual comparisons by ranking methods," in *Breakthroughs in Statistics*. Springer, 1992, pp. 196–202.

[64] N. Cliff, *Ordinal methods for behavioral data analysis*. Psychology Press, 2014.

[65] T.-Y. Lin, P. Goyal, R. Girshick, K. He, and P. Dollár, "Focal loss for dense object detection," in *Proceedings of the IEEE international conference on computer vision*, 2017, pp. 2980–2988.

[66] S. Casas, W. Luo, and R. Urtasun, "Intentnet: Learning to predict intention from raw sensor data," in *Conference on Robot Learning*, 2018, pp. 947–956.

[67] W. Wang, C. Wu, and M. Yan, "Multi-granularity hierarchical attention fusion networks for reading comprehension and question answering," in *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics, Volume 1: Long Papers*, 2018, pp. 1705–1714.

[68] W. Chen and J. Hays, "Sketchygan: Towards diverse and realistic sketch to image synthesis," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018, pp. 9416–9425.

[69] Z. Gao, X. Xia, D. Lo, J. Grundy, and T. Zimmermann, "Automating the removal of obsolete todo comments," in *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 1–12. [Online]. Available: https://arxiv.org/abs/2108.05846