This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2022.3185458



# Predictive Comment Updating with Heuristics and AST-Path-Based Neural Learning: A Two-Phase Approach

Bo Lin, Shangwen Wang, Zhongxin Liu, Xin Xia, and Xiaoguang Mao

**Abstract**—Just-in-time comment update is a promising way to reduce the burden of developers during software maintenance and evolution. Existing approaches can be divided into two categories: the heuristic-based approach and the deep-learning-based approach. The heuristic-based approach is restricted to a specific type of comment updates (i.e., code-indicative updates), but performs well on such type. The effectiveness of deep-learning-based approach is limited but it can handle diverse comment updates. Considering the complementary advantages of existing approaches, an intuitive idea is to combine them for better performance. To investigate this idea, we first conduct a pre-study experiment which shows that to construct an effective comment updater by combining heuristic-based and deep-learning-based approaches, we need to tackle two main challenges: 1) the heuristic-based approach may bring side effects to cases which cannot be updated by it; and 2) the current deep-learning-based approach is with limited effectiveness. Then, we propose a novel two-phase approach named `Toper` to cope with these two challenges and effectively perform comment updates. In the first phase, `Toper` integrates nine distinctive features identified through our large-scale empirical analysis into a predictive model, which can predict whether the contents of the comment updates can be found in the corresponding code changes, namely, the comment updates are code-indicative updates. If so, the updates are then generated by an off-the-shelf heuristic-based approach; otherwise, `Toper` leverages a deep learning model, which we specially designed for non-code-indicative updates, to infer the new comment based on the old comment and code change. Motivated by our manual observation on the limitation of existing approaches on non-code-indicative updates, our deep learning model adopts the Abstract Syntax Tree path technique, which can capture the program structure information for effectively embedding code changes. Our evaluation shows that our approach outperforms the state-of-the-art by around 20% with respect to the number of correct comments it generates. Via in-depth analysis, we illustrate the rationale of each design decision as well as point out potential directions.

**Index Terms**—Comment update, Deep learning, Code embedding.

## 1 Introduction

The natural-language comment is crucial for understanding the associated code [1], [2], [3], [4], [5], [6], [7]. Usually, the comments record information, such as the intention of the functionality as well as the implementation details, which can facilitate the communication among developers [2], [8], [9], [10], [11], [12], [13], [14]. Despite the significance of code comments for program comprehension, the comments are not always updated simultaneously with the accompanying source code during software evolution [15], [16], [17], [18]. As revealed by existing studies, such inconsistencies can lead to future project bugs [15] and thus hinder software maintenance activities [19], [20], [21], [22], [23].

In order to alleviate the inconsistency problem, recent studies have proposed diverse approaches to automatically update comments according to code changes, i.e., just-in-time (JIT) comment update [24], [25], [26]. Specifically, existing approaches can be classified into two categories, *heuristic-based approaches* and *deep-learning-based approaches*. Heuristic-based approach (e.g., *HebCup* [24]) is simple and effective. However, it can only work on comment updates whose changed contents can be found from the corresponding code changes (i.e., **code-indicative updates** [24]), and can not handle non-code-indicative updates. Deep-learning-based approaches (e.g., *CUP* [25]) are not restricted by update types theoretically. However, they are currently less effective than heuristic-based approaches on code-indicative updates and with respect to the overall performance [24]. In addition, despite some insightful explorations that have been performed towards this direction, the literature approaches are still far away from being applied in practice.

Considering that the advantages of heuristic-based approaches and deep-learning-based approaches seem to be complementary, an exciting idea is that can we combine the two categories of approaches to design better approaches for JIT comment update? Based on this idea, we performed a pre-study analysis by utilizing HebCup to update all the cases first and then referring to CUP for cases that cannot be handled by HebCup, but obtained unsatisfactory results. This pipeline is just as effective as the existing HebCup. Although it has not been investigated by prior works, we consider it as the state-of-the-art in JIT comment update because it combines the strengths of both the heuristic-based approach and the learning-based approach. Through our in-depth analysis, we identified two major limitations of this pipeline: 1) the lack of a predictor that can decide whether


- *Bo Lin, Shangwen Wang, and Xiaoguang Mao are with the National University of Defense Technology, China. E-mails: linbo19@nudt.edu.cn, wangshangwen13@nudt.edu.cn, and xgmao@nudt.edu.cn*
- *Zhongxin Liu is with Zhejiang University, China. E-mail: liu_zx@zju.edu.cn*
- *Xin Xia is with Huawei, China. E-mail: xin.xia@acm.org*
- *Shangwen Wang and Zhongxin Liu are the corresponding authors.*

a case should be fed to HebCup since non-code-indicative updates may be mistakenly updated by HebCup, and 2) the ineffectiveness of CUP on non-code-indicative updates since it can only correctly update 1.7% of such cases. In this paper, we propose `Toper`, a Two-phase cOmment uPdatER, to cope with the limitations of the state-of-the-art and move one step further towards this direction. `Toper` adopts a two-phase workflow where it first predicts to which type of update the target comment update instance belongs, and then for different update types it utilizes different ways to update the old comment. Considering the identified limitations, the main technical challenges we face in this study are ① how to accurately predict whether the content of the comment update will appear in the code change and ② how to effectively infer the comment update whose contents do not appear in the code change.

To address the first challenge, we performed an empirical analysis on 80,591 method-comment co-change samples, aiming to find out discriminative features between code-indicative and non-code-indicative updates with respect to code changes. We analyzed the features of each code change from three aspects, i.e., the complexity of the code change, the extent to which the modified code occurs in the old comment, and the context information of the code change, respectively. Totally, we found nine features that can effectively help us distinguish the two types of comment updates. To address the second challenge, we also made an in-depth manual analysis concerning the factors that affect the performance of the current technique (i.e., CUP) on 6,234 non-code-indicative updates. We found that the most significant one is that CUP often fails to predict the comment contents that need to be updated. Our further observation is that code changes that (1) happen under similar program structural contexts and (2) cast the same operations (e.g., insertion, deletion, and replacement) on code tokens may result in the updates of comment tokens within similar natural language contexts. As the concrete example presented in Fig. 3, two instances with code changes in similar program structural update the corresponding comment in similar locations. Detailed illustration will be shown in Section 4.2.2. Such phenomenons indicate that incorporating the structure information of code change may help decide which comment tokens should be updated. Considering the state-of-the-art comment updater, CUP, ignores such structure information, we are motivated to consider such information when updating non-code-indicative ones.

Therefore, given an instance (a code change plus with an original comment), `Toper` works as follows: First, it constructs a classifier to predict if the instance is a code-indicative update based on the aforementioned nine code change features. If the classifier identifies the comment update as code-indicative, `Toper` directly reuses *HebCup* to update the comment, since *HebCup* effectively addresses such situations. If not, we send the instance to our specially designed deep learning model, which is expected to predict the update results well. Our model, in general, follows the encoder-decoder paradigm. Different from CUP, it involves the structure information of the code change within the encoder. Specifically, we utilize the paths that connect adjacent Abstract Syntax Tree (AST) nodes to represent the program structure information during encoding (such a technique is known as the *AST path* [27], [28]). Furthermore, we incorporate the code change operation on the leaf node into each path to make the model be aware of how the code is changed. After obtaining the whole bag of paths for the changed method, an attention mechanism is used to integrate the embeddings of all paths and represent the code change. For the old comment, we still treat it as a token sequence. After separately representing the code change and the old comment as feature vectors, we concatenate the two vectors and send the result to the decoder, where the predicted new comment is generated.

We performed extensive experiments to demonstrate the effectiveness of our proposed approach on the existing carefully curated dataset provided by Lin et al. [24], which contains a total of 98,622 change instances of Javadoc comments with its associated code. Our results show that (1) compared with the state-of-the-art pipeline, heuristic-based HebCup, and learning-based CUP approaches, our `Toper` can generate about 20%, 20%, and 90% more correct comments (i.e., those are identical to the developer-provided ones) respectively on the whole test set; (2) our classifier can generally work well, with both accuracy and F-score exceeding 80%; (3) our AST-path-based comment updater can generate three times as many correct comments as CUP on the non-code-indicative updates; and (4) the two key components of our approach (i.e., the classifier and the AST-path-based updater) contribute together to the overall effectiveness. Through further in-depth case studies, we point out potential directions for future studies towards just-in-time comment update.

To sum up, our study makes the following contributions:

- Our empirical study deepens the understanding towards the difference of naturalness between code-indicative and non-code-indicative updates. Based on our empirical findings, we design a classifier that can determine whether an instance is a code-indicative update based on nine code change features.
- Our investigation exposes the limitation of current techniques on updating non-code-indicative ones. Based on that, we design an AST-path-based comment updater that takes into consideration the program structure information for updating non-code-indicative ones.
- Built on top of the proposed classifier and AST-path-based updater, we implement a two-phase comment updater, `Toper`, which adopts different strategies to deal with code-indicative and non-code-indicative updates. The `Toper` is open-source to facilitate future just-in-time comment update studies at: **https://zenodo.org/record/5340569**.
- We perform extensive experiments to assess the performance of `Toper`. Results reveal that `Toper` builds a more advanced baseline for future studies and every design decision makes sense.

## 2 Background

This section presents the necessary background knowledge about our study, including the definitions of different types of comment updates, the AST related concepts, as well as the advantages and disadvantages of existing just-in-time comment update approaches.

## 2.1 Definitions

### 2.1.1 Alignment

A critical step for code change and comment update analysis is alignment, which can reflect how the code (or comment) is modified [24], [25]. Given a code change instance where the token sequence of the old code is $s = t_1, t_2, \ldots, t_j$, and the token sequence of the new code is $s' = t'_1, t'_2, \ldots, t'_k$ (i.e., the original lengths of the two sequences may not be identical), the alignment result is represented as an edit sequence $(< t_1, t'_1, o_1 >, \ldots < t_n, t'_n, o_n >)$. $t_i$ is a token in the old code, $t'_i$ is a token in the new code, and $o_i$ is an operation converting $t_i$ to $t'_i$ which could be *UPDATE*, *DELETE*, *INSERT*, or *KEEP*. If $o_i$ is *INSERT* (*DELETE*), $t_i$ ($t'_i$) will be the empty token $\phi$. Note that the above example is just a showcase. In practice, the alignment can be applied (1) at token or sub-token level (i.e., in this study, all tokens are split into sub-tokens based on the camel case and underscore naming conventions and then transformed into lowercase, following previous studies [27], [29]) and (2) for code changes or comment updates.

**Token replacement pair.** From the alignment results, for the triple $< t_i, t'_i, o_i >$, if $t_i \neq t'_i$, then $< t_i, t'_i >$ is called a *token replacement pair*. For two token replacement pairs $< t_i, t'_i >$ and $< t_j, t'_j >$, if $t_i = t_j$ and $t'_i = t'_j$, then these two pairs are *redundant*.

### 2.1.2 Comment update classification

In our study, we categorize comment updates according to whether their updated contents can be explicitly found in the corresponding code change contents. Note that this classification standard was initially proposed by Lin et al. [24]. Here we give the formal definition as below. Given a code change with its corresponding comment update, suppose the aligned sub-token level edit sequence of the code change is:

$$(< STcd_1, ST'cd_1, Ocd_1 >, \ldots, < STcd_x, ST'cd_x, Ocd_x >)$$

Similarly, the sub-token level edit sequence of the corresponding comment update is:

$$(< STcm_1, ST'cm_1, Ocm_1 >, \ldots, < STcm_y, ST'cm_y, Ocm_y >)$$

If for any element $e$ $(< STcm_e, ST'cm_e, Ocm_e >)$ in the second edit sequence whose $Ocm_e \in \{\texttt{UPDATE}, \texttt{DELETE}, \texttt{INSERT}\}$, there always exists an element in the first sequence $(< STcd_l, ST'cd_l, Ocd_l >)$ where $STcd_l = STcm_e$, $ST'cd_l = ST'cm_e$, and $Ocd_l = Ocm_e$, then this instance is a **code-indicative update**. That is to say, if all the updated contents of the comment can be found from the corresponding code change, it is considered as a code-indicative comment update. Otherwise, it is a **non-code-indicative** update. Generally speaking, code-indicative updates are about cases where renaming happens in the source code and a mention of the renamed object in the comment must be correspondingly updated with a *textually identical* edit. This *textually identical* property makes heuristic-based approach effective on such type of comment updates, whose workflows will be detailed later.

A concrete example of code-indicative update is shown in Fig. 1. In this example, the only element from the aligned sub-token level edit sequence of the comment change whose operation $\in \{\texttt{UPDATE}, \texttt{DELETE}, \texttt{INSERT}\}$ is $< \phi, update, \texttt{INSERT} >$ (note that in this study, tokens will be split into sub-tokens and then transformed into lowercase. Hence, the updated token `updateVersion` is split into sub-tokens `update` and `version`). Also, there is an identical element in the aligned sub-token level edit sequence of the code change. Therefore, this example is a **code-indicative** update since it satisfies the pre-defined rules.

### 2.1.3 AST related concepts

**AST.** The AST of a code snippet is defined as a tuple: $\langle N, L, T, r, \Delta, \Phi \rangle$, where $N$ is a set of non-leaf nodes, $L$ is a set of leaf nodes, and $T$ is a set of code tokens for $L$. $r \in N$ represents the root node, $\Delta$ and $\Phi$ denote two sets of mapping functions. Specifically, $\delta \in \Delta : n \rightarrow n', n \in N, n' \in (N \cup L)$ is a function that maps a non-leaf node to its children nodes. $\phi \in \Phi : l \rightarrow t, l \in L, t \in T$ maps a leaf node to the corresponding code token.

**AST path.** An AST path is a path from the root node $r$ to a leaf node $l$, that is defined as a quadruple: $p = \langle r, l, N', \Delta' \rangle, l \in L, N' \subset N, \Delta' \subset \Delta$.

**Operation Path.** Generally, an operation path is an AST path associated with code change operator that works on a leaf node $l$ [30], which is defined as a triple: $op = \langle t, p, o \rangle$, where $t$ is the code token of the leaf node $l$ in the AST path $p$, and $o \in \{\texttt{DELETE}, \texttt{INSERT}, \texttt{KEEP}, \texttt{UPDATE}\}$ is an atomic code change operator that works on the leaf node. Specially, if the operation on a leaf node is `UPDATE`, the operation path will be $op = \langle t_{old}, p, t_{new} \rangle$. We do not assign a change operator for *update* since the embedded vectors of the new token and the operator will be significantly different and thus it can be inferred from the embedded results that it is an update. Considering the code change example in Fig. 1, the AST path of the red dotted line is $p = \langle MethodDeclaration \rightarrow Block \rightarrow ReturnStmt \rightarrow FieldAccessExpr \rangle$ while its operation path is $op = \langle version, p, updateVersion \rangle$.

## 2.2 Automated Comment Update

Researchers have formulated the task of automatically updating an existing comment when the corresponding body of code is modified. CUP [25] is such an approach that implements a generic neural sequence-to-sequence (seq2seq) model to learn comment update patterns. It represents the code changes as an edit sequence and integrates a novel co-attention mechanism that can effectively link and fuse information in code changes and comments. The advantage of CUP is that its application is not restricted by the type of comment update (i.e., it can work for both code-indicative and non-code-indicative updates), while the disadvantage is that its overall effectiveness is comparatively low [24], [25] (i.e., only capable for addressing 16% of the total cases). Panthaplackel et al. [26] also utilized a sequence to represent code change and train their model to generate a sequence of edit actions to apply to the comment. Nevertheless, their approach was only evaluated on Javadoc comments beginning with `@return` while in this study we use a dataset which consists of different types of Javadoc comments. Unlike the previous two approaches, HebCup [24] is based on a set of specially-designed heuristics. Since it is a component of our proposed approach, we briefly recall how it works here.

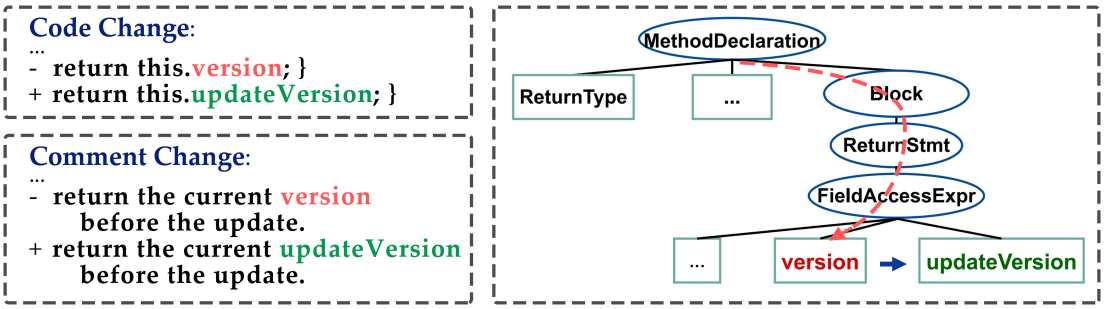



**Fig. 1:** An instance of code-indicative update.

Given a code change, it first finds out code tokens that are changed. It then uses an alignment technique to identify the modified sub-token in each changed token. After that, HebCup constructs token-level replacement pairs, which enumerate all possible old tokens with their potential new tokens. Finally, HebCup traverses tokens in the old comment and any of them that matches the *old tokens* in the replacement pairs will be updated with the *potential new tokens* in the replacement pairs. Therefore, the output of HebCup is determined. The advantage of HebCup is that it works quite effectively (i.e., outperforming CUP to a large extent [24]), while the disadvantage is that its success requires a precondition that the updated content of the comment can be explicitly found in the code change (i.e., HebCup can only generate correct comments for code-indicative updates). It should be noted that HebCup will make changes to any old comment once the comment contains tokens that appear in the code changes. For any instance where there is no common token between the old comment and the code change, HebCup outputs the old comment directly. That is to say, HebCup will also produce a comment even if the instance is not a code-indicative one. However, such comments are not identical to the ground truth since the oracle comment changes are not reflected by the code changes.

It should be noted that there are some other works that hold the potential to deal with comment updates. For instance, traditional code summarization techniques [31], [32] could be applied to directly generate comments for the updated code although doing so would ignore the code change information, which is important for guiding comment updates. DRONE [33], the detection and repair tool for API documentation defects, may also be used to recommend updates to specific types of old comments despite that its generality to code comments remains unknown. To the best of our knowledge, the three studies introduced in the last paragraph are the only ones that explicitly tackle the JIT comment update task so far. Therefore, we consider them as our study subjects.

## 3 The State-of-the-Art Pipeline

Considering the complementary advantages of heuristic-based (i.e., HebCup) and deep-learning-based (i.e., CUP) comment updaters, an intuitive idea is that we can combine two approaches to achieve better performance on JIT comment update. A straightforward pipeline is to use HebCup to generate updates first and refer to CUP for those that cannot be resolved by HebCup. Concretely, if all the tokens in the old comment do not appear in the code change, HebCup will make no update and return the old comment, and we consider such cases as unresolved by HebCup. This pipeline can utilize both the effectiveness of HebCup on dealing with code-indicative updates and the capacity of CUP on performing non-code-indicative updates and hence is considered as the state-of-the-art pipeline in this paper. Note that we determine whether Hebcup has updated the comment by judging the consistency of the input and output comments.

**TABLE 1:** Performances of the state-of-the-art comment updater.

| Approach | Accuracy |
|---|---|
| CUP | 15.8% |
| HebCup | 25.6% |
| HebCup + CUP | 25.8% |

In this section, we perform a pre-study experiment by analyzing the effectiveness and the limitations of the straightforward pipeline mentioned above to comprehensively understand the state-of-the-art pipeline of JIT comment update.

### 3.1 Effectiveness

To perform the pre-study experiment, we use the dataset constructed by Liu et al. [25] and further distilled by Lin et al. [24], which contains 80,591/8,827/9,204 comment update instances for training/validation/test. Since the evaluation results of CUP and HebCup on the test set have already been provided by Lin et al. [24], we can directly compare the effectiveness of the proposed pipeline with them.

We adopted the open-source implementations of CUP and HebCup. CUP was trained and validated following the original paper [25], and then tested on the test set. As for comparison, HebCup does not need a training process and can be directly applied. In this pre-study analysis, we focused on the **Accuracy** metric, which assesses the percentage of the test samples where the correct comments (i.e., comments identical to those provided by developers) can be generated. Results are shown in Table 1.

From the results, we note that the accuracy of the state-of-the-art is only slightly higher than that of HebCup (25.8% vs. 25.6%). This result indicates that simply combining HebCup with CUP does not result in significant improvements.

### 3.2 Limitation

To understand why the state-of-the-art does not outperform the existing techniques significantly, we manually analyzed all the 6,830 cases where correct comments are not generated

**TABLE 2:** A non-code-indicative instance updated by HebCup.

| |
|---|
| **Code Change**:<br>public static ModuleIdentifier asModuleIdentifier(ModelNode value) {<br>− return optionalModuleIdentifier(value).Else(null);<br>+ return optionalModuleIdentifier(value.asString());<br>} |
| **Old Comment**: Returns the value of the node as a module identifier, or nothing if the node is undefined. |
| **New Comment**: Returns the value of the node as a module identifier. |
| **HebCup**: Returns the value.asString of the node as a module identifier, or nothing if the node is undefined. |

by the pipeline. We mainly focused on two aspects during the analysis: 1) if there is any case that can be correctly updated by an individual approach but is incorrectly updated by the pipeline; and 2) if there is a significant difference between the effectiveness of the pipeline on code-indicative and non-code-indicative updates respectively. Our analysis shows that both the questions are confirmed, and through in-depth analysis we summarized two key reasons for the unsatisfactory performance of the state-of-the-art.

First, an instance that can be correctly updated by CUP may be incorrectly updated by the pipeline, since HebCup may mistakenly update non-code-indicative instances. Recall the workflow of HebCup we have introduced in Section 2.2, HebCup will update tokens in the old comment as long as they are identical to the tokens changed in the code change. However, in non-code-indicative updates, such matching relations may also exist but we do not need to update the corresponding comment tokens. Under such conditions, HebCup will perform incorrect modifications and these instances will not be sent to CUP later, leading to the generation of incorrect comment updates. Table 2 shows a case from the test set. In this non-code-indicative instance, developers update a `return` statement by changing `value` to `value.asString()` and remove the invocation of `Else(null)`. The deletion of `Else(null)` significantly changes the semantic of this method. Therefore, developers remove the description about this method invocation (e.g., , "or nothing if the node is undefined") in the corresponding comment. HebCup, on the contrary, constructs the token replacement pair `value` → `value.asString` after alignment. It thus mistakenly updates the comment of this non-code-indicative instance, as shown in the table. We also note that this example can be correctly handled by CUP. Therefore, this example shows that by simply using HebCup first and CUP later, the straightforward pipeline may prevent some instances being correctly updated by CUP. Totally, this is the case that happens for 136 instances in the test set, which is a non-negligible number considering that CUP can only correctly update 1,456 instances. The root cause of this limitation is that we do not know whether an instance is a code-indicative or not without the ground truth. If there is a perfect classifier which can provide such information, this limitation can be easily mitigated. Therefore, our first finding suggests that to effectively combine heuristic-based and deep-learning-based approaches, we need to precisely predict whether a comment update is a code-indicative one.

Second, the effectiveness of the pipeline on code-indicative updates is significantly better than that on non-code-indicative ones, since CUP is generally ineffective at dealing with non-code-indicative updates, which has already been observed and analyzed by Lin et al. [24]. Results reveal that the pipeline can only generate correct updates for 94 out of the 6,234 non-code-indicative updates in the test set with an accuracy of 1.7%. On the contrary, its accuracy on code-indicative updates is 45.9% (1,362/2,970). Therefore, our second finding suggests that to perform the JIT comment update effectively, we need to improve the effectiveness of the approaches on non-code-indicative updates.

**TABLE 3:** Features investigated in our empirical study.

| Dim | Feature | Definition |
|---|---|---|
| Complexity | NMS | Number of modified sub-tokens |
| | NMT† | Number of modified tokens |
| | NML | Number of modified lines |
| | NMC | Number of modified chunks |
| | NNTRP | Number of non-redundant token replacement pairs |
| | NNSRP | Number of non-redundant sub-token replacement pairs |
| Involvement | NTOD | Number of tokens which occur in the old comment but disappear after the code change |
| | NSOD | Number of sub-tokens which occur in the old comment but disappear after the code change |
| Context | TS | The type of the statement where the changed token locates |
| | TE | The type of the expression where the changed token locates |

† $NMT$ in this paper denotes Number of Modified Tokens. It has no correlation with the abbreviation of neural machine translation.

## 4 Empirical Study

Through our pre-study analysis, we identified the main challenges that need to be addressed by an effective comment updater are (1) accurately determining if a comment update is a code-indicative one, and (2) effectively inferring comment updates for non-code-indicative ones. In this section, we provide the empirical evidence to support the design strategy of our approach for addressing such challenges.

### 4.1 Discriminative Features Between Code-Indicative and Non-Code-Indicative Updates

#### 4.1.1 Studied features

In this part, we aim to investigate the question *which code change features can be utilized to distinguish code-indicative and non-code-indicative updates*. To achieve so, we analyzed comment update instances to see if the two types of updates tend to possess different values with respect to any feature of the corresponding code change. We analyzed the features of code change from three dimensions, which are *code change complexity* indicating the complexity of the code change, *code change involvement* indicating the extent to which the modified contents of the code change occur in the old comment, and *code change context* indicating the program context of the changed code. Table 3 summarizes the ten features that we investigated. Please note that all the features in this paper are quantitatively analyzed for the first time under the comment update context.

The reasons we investigated complexity features of the code change are (1) the previous study [24] has shown that code-indicative updates are more prone to appear in simple comment updates (e.g., the update whose changed content is only related to one token) and (2) the complexity of a comment update may be positively correlated with

the complexity of the associating code change. Therefore, simple code changes may tend to result in code-indicative updates. The first four features measure how many sub-tokens are changed *(NMS)*, how many tokens are changed *(NMT)*, how many code lines are changed *(NML)*, and how many code chunks are changed *(NMC)*, respectively. Note that a changed code chunk is a sequence of code statements that are continuously changed. These features measure the code change complexity from different granularities and are thus widely used in the previous studies [34], [35], [36], [37], [38], [39], [40]. The fifth and sixth features focus on the alignment results, measuring the number of non-redundant token replacement pairs *(NNTRP)* and the number of non-redundant sub-token replacement pairs *(NNSRP)*, respectively. The intuition behind it is that purely relying on NMT and NMS may not reflect the code change complexity comprehensively. For instance, if the token `A` is changed into `B` 10 times in a code change, the NMT will be 10. However, the code change is only related to the replacement of one token. Under this situation, the NNTRP will only be 1, therefore, reflecting the code change complexity from another perspective.

Two features are designed for the *code change involvement* dimension: *number of tokens which occur in the old comment but disappear after the code change (NTOD)* and *number of sub-tokens which occur in the old comment but disappear after the code change (NSOD)*. Here, we refer to *disappeared* as the token (or sub-token) occurs in the old code but does not occur in the new code (i.e., it can be both removed and replaced during the code change). The intuition for investigating these features is that if the *disappeared* token is within the old comment, it is likely that the comment should be updated based on the code change content of this token and the probability increases when the number of this type of tokens increases (the same for sub-token).

Regarding the *code change context* dimension, we investigated two features for each changed code token, which are the type of the statement *(TS)* where it locates (e.g., If statement and Return statement) and the type of the expression *(TE)* where it locates (e.g., Infix expression and Assignment). Here, we refer to *changed* as updated, inserted, or deleted. The intuition is that tokens under diverse program contexts may contribute unequally to the program comment, inspired by previous works on program comprehension [29], [41]. Therefore, we postulated that the comment is more likely to be updated according to the modifications of code tokens under specific program contexts.

#### 4.1.2 Empirical analysis

Liu et al. [25] created a large-scale and open-sourced dataset to evaluate the performance of CUP. Recently, Lin et al. [24] removed the noisy data (i.e., instances where the comment updates only optimize the language expression while the semantics remain identical, such as typo fixings) in this dataset. Therefore, we chose to reuse the cleaned dataset in this study. Specifically, the cleaned dataset contains 80,591, 8,827, and 9,204 method-comment co-change samples for training, validation, and test sets, respectively. In the following, we performed an empirical study on the training set to confirm if our considered features can help distinguish code-indicative and non-code-indicative updates. It should be noted that the label of whether an instance is code-indicative or not has been already provided by Lin et al. [24].

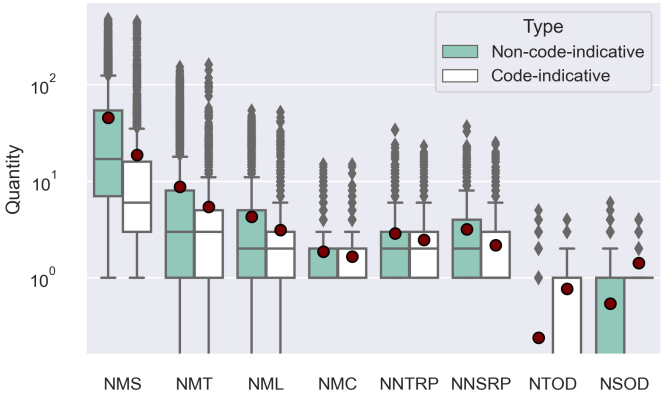



**Fig. 2:** Value distributions of the eight features.

**TABLE 4:** P-values and Cliff's delta for the eight features in the complexity and involvement dimensions comparing *code-indicative* and *non-code-indicative* updates on the training set.

| Feature | P-value | Cliff's delta |
|---|---|---|
| NMS | $< 0.001$ | -0.37 (Med) |
| NMT | $< 0.001$ | -0.03 (Negligible) |
| NML | $< 0.001$ | -0.07 (Negligible) |
| NMC | $< 0.001$ | -0.05 (Negligible) |
| NNTRP | $< 0.001$ | -0.08 (Negligible) |
| NNSRP | $< 0.001$ | -0.26 (Small) |
| NTOD | $< 0.001$ | 0.45 (Med) |
| NSOD | $< 0.001$ | 0.54 (Large) |

**Code change complexity & Code change involvement.** The distributions of the values of the eight features concerning *code change complexity* and *code change involvement* are illustrated in Fig. 7. From the results, the distributions of values of all the features differ across different types of updates. For instance, for the *NMS*, the median value of non-code-indicative updates is 17, significantly higher than that of code-indicative updates which is 6. For another four features (i.e., NMT, NML, NMC, and NNTRP), while the two types of updates possess identical median values, the average values of non-code-indicative updates are still higher than those of code-indicative ones to a non-negligible extent (e.g., 1.86 vs. 1.65 with respect to *NMC*). As for the two *code change involvement* features (i.e., NTOD and NSOD), it is obvious that the average values of code-indicative updates are higher than those of non-code-indicative ones.

For each feature, we further performed a one-sided Mann-Whitney U-Test [42] to analyze the statistical significance of the difference between code-indicative and non-code-indicative updates. We also computed Cliff's delta [43], a non-parametric effect size measure that can evaluate the amount of difference between two variables. [1] Results are shown in Table 4, Besides, the distribution differences are all statistically significant (i.e., p-value $< 0.001$) for the eight features, indicating that the features are generally effective in differentiating code-indicative updates from non-code-indicative ones. With respect to the Cliff's delta, four features have negligible effect sizes (i.e., NMT, NML,

1. Cliff defines a delta of less than 0.147, between 0.147 and 0.33, between 0.33 and 0.474 and above 0.474 as negligible, small, medium, large effect size, respectively.

NMC, and NNTRP), one feature has a small effect size (i.e., NNSRP), two features have medium effect sizes (i.e., NMS and NTOD), and one feature has large effect size (i.e., NSOD). Features with negligible effect sizes cannot be simply considered as worthless for determining code-indicative updates. They may be weak indicators but combining these features with features from other dimensions may improve the performance of our classifier (as we will illustrate in Section 6.4).

**Code change context.** We investigated whether the contexts under which the code changes tend to trigger code-indicative updates are different from those contexts tending to trigger non-code-indicative updates. Table 5 illustrates the top-5 contexts where the code changes of code-indicative and non-code-indicative updates happen and their corresponding proportions. All expression and statement types are parsed by the *javalang* package [2] and we use *MethodDeclaration* to denote the method signature. A full list of all involved expression and statement types can be found at the Eclipse official documentation. [3]

Totally, we found 607,906 changed code tokens from all the instances, where 129,258 and 478,648 come from code-indicative and non-code-indicative updates, respectively. Regarding the expression type, the distributions of contexts of changed code tokens for code-indicative and non-code-indicative updates tend to be similar. For instance, for both update types, the majority occurs in the `Name` expression, occupying around 85% of the total number. The proportion differences of other expression types are also insignificant. We also calculated the proportion of the tokens in the `Name` expression to all the code tokens in the dataset, which turned out to be around 80% (1,807,026/2,333,758). This shows that the skewed data (i.e., 85%) does not necessarily mean that code changes tend to appear in the `Name` expression.

When it comes to the statement type, the differences between the two types of updates become apparent. For example, 26.3% changed code tokens for code-indicative updates occur in `MethodDeclaration`, while this proportion for non-code-indicative updates is 12.4%. Such a result indicates that if a token under the `MethodDeclaration` context is changed, the comment update is more likely to be a code-indicative one.

**Correlations and importance of features.** Following Fan et al. 's study [44], we investigate the possible associations between features and apply feature selection, aiming to remove correlated features that might lead to poor models [45]. Note that we only investigate the eight features from the code change complexity and involvement dimensions, because the features in the code change context are the statement and expression types rather than numerical features.

*S*tep 1: Correlation Analysis. To reduce the bias of the model and avoid the correlated features, we first utilize the variable hierarchical cluster analysis to look for the correlations among features, which is implemented as **varclus** function in the R package **Hmisc**. The **varclus** constructs a hierarchical overview of the features and groups the correlated features into sub-hierarchies. Following the previous study [46], we consider the features are correlated if the correlations of features in the sub-hierarchy are above 0.7. The results show that none of the features are strongly correlated. The most correlated features are NML and NMT with a correlation of 0.64, which is still lower than 0.7.

*S*tep 2: Redundancy Analysis. After checking the collinearity among the features by correlation analysis, we then apply redundancy analysis by the **redun** function in the R package **Hmisc** to determine how well each feature can be predicted from the remaining features. Through the redundancy analysis, we find that none of the eight features are redundant.

*S*tep 3: Feature Importance. We investigate the contribution of each feature group by performing an ablation study where we removed features in one group at a time and retrained our classifier based on the rest features and re-assessed its effectiveness. Note that from the observation of the previous steps, no features are strongly correlated or redundant, so there is only one feature per group and we remove one feature at a time. The result shows that the three most important features ranked by the degradation of F1-score are NSOD, NMS, and NNTRP. The detailed analyses of each feature are in Section 6.4.

*Recall that we totally investigated ten features for the code change. Code-indicative and non-code-indicative updates demonstrate distinctions on nearly all of them, except for the expression type context.*

## 4.2 Features that can be Leveraged to Infer Non-Code-Indicative Updates

### 4.2.1 Weakness of current technique

In this part, we aim to investigate the question of *how to infer non-code-indicative updates better*. To achieve this goal, we first need to understand the weakness of the existing approach on these updates. We thus analyzed the performance of CUP on non-code-indicative updates within the aforementioned test set. Such data is already provided by the previous study [24]. Note that this investigation is made via manually comparing the predicted comments from CUP and the oracle comments. Two authors independently observed the symptoms of CUP's failures, and they then reached a consensus about the representative cases via a discussion. The workload of each participated author is 6,234 cases.

Through our manual analysis, we found the most significant factor that limits the performance of CUP on the non-code-indicative updates is that CUP tends to perform updates at incorrect locations (i.e., updating tokens that do not need to be updated). Specifically, among all the 6,234 non-code-indicative updates in the test set, there are a total of 8,201 tokens that need to be updated according to the oracle updates from developers. CUP only modifies 1,485 of them while simultaneously updates 4,708 tokens that do not need to be updated.

We observed that such inaccuracy of locating where to perform the update leads to the ineffectiveness of CUP on non-code-indicative updates. An example is given in Table 6. The instance is a non-code-indicative update since the updated token, *milliseconds*, does not appear in the code. CUP does capture the semantic meaning of the

2. https://github.com/c2nes/javalang
3. https://help.eclipse.org/latest/index.jsp

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2022.3185458



**TABLE 5:** The distributions of the contexts of code changes from code-indicative & non-code-indicative updates.

| | Code-Indicative | | | Non-Code-Indicative | | |
|---|---|---|---|---|---|---|
| | **Type** | **Quantity** | **%** | **Type** | **Quantity** | **%** |
| Expression | Name | 111,071 | 85.93 | Name | 400,628 | 83.7 |
| | InfixExpression | 5,829 | 4.51 | InfixExpression | 25,894 | 5.4 |
| | StringLiteral | 5,402 | 4.18 | StringLiteral | 20,677 | 4.3 |
| | NumberLiteral | 2,817 | 2.18 | NumberLiteral | 11,870 | 2.5 |
| | Assignment | 1,951 | 1.51 | Assignment | 8,663 | 1.8 |
| | Others | 2,188 | 1.69 | Others | 10,916 | 2.3 |
| | Total | 129,258 | 100.0 | Total | 478,648 | 100.0 |
| Statement | StatementExpression | 38,245 | 29.6 | StatementExpression | 140,060 | 29.3 |
| | MethodDeclaration | 33,978 | 26.3 | LocalVariableDeclaration | 117,076 | 24.5 |
| | LocalVariableDeclaration | 22,765 | 17.6 | MethodDeclaration | 59,295 | 12.4 |
| | ReturnStatement | 16,226 | 12.6 | IfStatement | 60,668 | 12.7 |
| | IfStatement | 10,207 | 7.9 | ReturnStatement | 56,300 | 11.8 |
| | Others | 7,837 | 6.1 | Others | 45,249 | 9.4 |
| | Total | 129,258 | 100.0 | Total | 478,648 | 100.0 |

**TABLE 6:** An incorrect comment update from CUP.

| |
|---|
| **Code Change**:<br>− public static Instant getInstantMillisOffsetFromNow(long offsetInMillis) {<br>+ public static Instant getInstantHoursOffsetFromNow(long offsetInHours) {<br>− return Instant.now().plus(Duration.ofMillis());<br>+ return Instant.now().plus(Duration.ofHours());<br>} |
| **Old Comment**: Returns an java.time.Instant object that is offset by a number of milliseconds from now. |
| **New Comment**: Returns an java.time.Instant object that is offset by a number of hours from now. |
| **CUP**: Returns an java.time.Hours object that is offset by a number of milliseconds from now. |

code change since it tries to update the comment by including the newly-added sub-token from the code change (i.e., , *Hours*). Nonetheless, without realizing where to perform the change, it updates a wrong comment token (java.time.Instant → java.time.Hours) and thus leads to this failure. Such an example demonstrates that comparing with generating the content that is used to update the comment, predicting the location where to perform the update is a more difficult problem for CUP. We, therefore, gain the observation:

*The current technique (i.e., CUP) is mainly challenged by predicting the comment tokens that need to be changed for non-code-indicative updates.*

### 4.2.2 Our observation

From the above analysis, we further investigated *whether there is any feature that can help predict the comment tokens to be updated*. Fig. 3 presents two instances with similar code changes and comment update locations. In Fig. 3a, we demonstrate the code change, the comment update, and the program structural context of the changed code, all of which are from the aforementioned instance in Table 6. In Fig. 3b, we give the identical information for another instance from the training set. To illustrate the program structure information, we adopt a widely-used concept, the Abstract Syntax Tree (AST) path [27], [28], which has been introduced in Section 2.1. In such a path, a code token is connected with the root node by a number of its ancestor nodes in the AST. For instance, in the red dotted path of Fig. 3a, the node `ReturnStmt` is connected with the root node `MethodDeclaration` through its parent node `Block`.

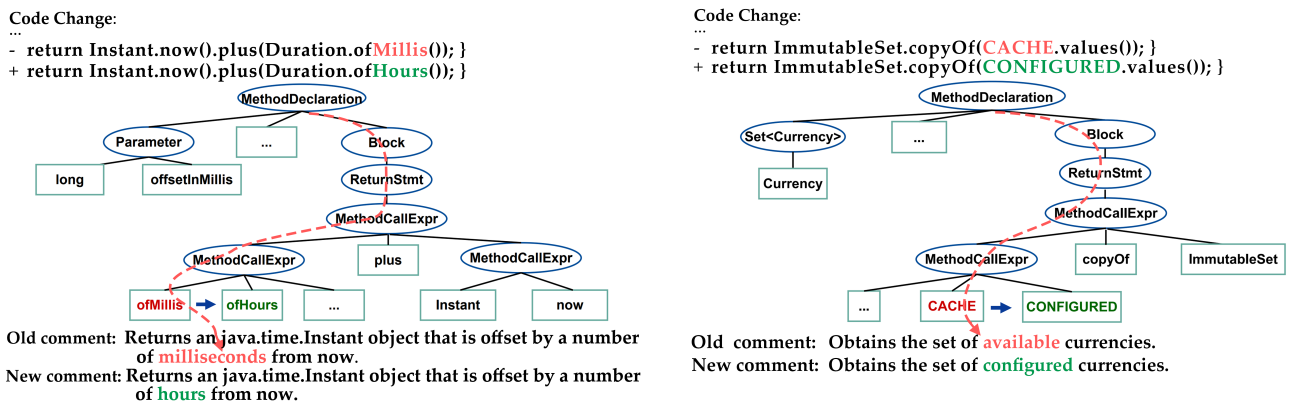
From this example, we gain the following observations. First, the changed codes in these two examples are under a similar program structural context: both are under a method call expression within a `return` statement. Their AST paths can also reflect this: as illustrated by the red dotted lines, all the non-leaf nodes in the two paths are identical. Second, the code change types (e.g., addition or deletion) of the two instances are the same: both of them update one code token to another. Third, the comment update locations of the two instances are similar, both in a noun phrase after the preposition `of`. Therefore, it inspires us that *the same operation on code tokens under similar structural contexts may lead to similar modification locations in the comments*. Current techniques, neither CUP nor the one proposed by Panthaplackel et al. [26], take into consideration the structural context of code change since they represent code change only by token sequence. They thus miss the opportunity to predict where to perform the comment update precisely. Motivated by our observation, we propose to utilize AST path information for code change representation. Thus, we can alleviate this problem significantly, and also use pointer network to implicitly decide the comment locations requiring update (cf. Section 5.2.2). In our evaluation, by learning from the training instances like the one shown in Fig. 3b, `Toper` successfully captures the structural features of this code change and correctly updates the comment of the instance in Fig. 3a, while CUP fails to do so.

*The structural context of the code change couple with the code change type may help determine the location of the corresponding comment update.*

## 5 Methodology

In this section, we introduce our `Toper` in detail. The overall framework of our two-phase approach is shown in Fig. 4.

1) Given a code change plus with the old comment of the code, `Toper` first extracts several features of the code change and then sends the features into a classifier which will automatically predict whether the input is a code-indicative update.




**(a)** An instance from our test set.

**(b)** An instance from our training set.

**Fig. 3:** Two instances with similar code changes and comment update locations.

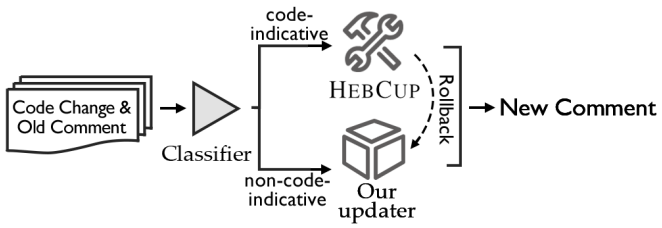



**Fig. 4:** Overview of our approach.

2) If the input is identified as a code-indicative update, `Toper` directly reuses HebCup for comment update.
3) Otherwise, `Toper` utilizes our newly proposed model, which is specially designed for non-code-indicative updates.
4) If HebCup cannot generate any update, we switch to the newly proposed model to update the comment.

### 5.1 Key Component#1: Code-Indicative Update Classifier

For the classifier, we use the first nine features in Table 3, which have been identified as helpful in differentiating different types of updates in Section 4.1. Each of the first eight features (i.e., NMS, NMT, NML, NMC, NNTRP, NNSRP, NTOD, and NSOD, respectively) is represented as an integer. As for TS, in case multiple code tokens are involved in a code change, we empirically decide to include the TS feature for the first three changed tokens. We performed a pre-study experiment to consider one, three, and five tokens separately and trained/tested the classifier on the training/testing set respectively. We found that three is the best choice. Our statistics show that for comment updates in the training set, the median value of changed tokens in the code change is 2 and only less than 40% of them change more than 3 tokens. We thus consider taking 3 tokens into consideration can capture enough context information. Also, such a number can avoid that many dimensions in the feature vector are related to this feature, under which condition the classifier may be excessively affected by this single feature. If the number of changed tokens is less than three, we pad the corresponding dimensions with zero. Therefore, the last three dimensions in the feature vector are used to denote the *code change context*, leading to the whole feature vector being 11-dimensional.

We experimented with four widely-used machine learning models: Decision Table [47], Naive Bayes [48], Logistic Regression [49], and Random Forest [50]. As shown in Table 7, the optimal results were obtained with Random Forest. Indeed, it has been shown to perform well in various classification tasks in software engineering [51], [52], [53], [54].

**TABLE 7:** Evaluation of classification performance on four ML classifiers.

| Classifier | Precision | Recall | Accuracy | F-score | AUC |
|---|---|---|---|---|---|
| RandomForest | **94.7** | 78.7 | **82.3** | **86.0** | **91.8** |
| DecisionTree | 91.1 | **79.9** | 81.1 | 85.2 | 85.3 |
| LogisticRegression | 94.7 | 68.7 | 76.2 | 79.6 | 86.3 |
| NaiveBayes | 89.1 | 29.9 | 50.0 | 44.8 | 80.3 |

### 5.2 Key Component#2: Non-Code-Indicative Comment Updater

Fig. 5 illustrates the overall framework of our updater, which follows the encoder-decoder paradigm. In the encoding step, given a code change with its old comment, we separately embed the code change and the comment and then concatenate these two vectors for the following step. In the decoding step, we generate the sub-tokens of the new comment one by one. Details of our updater are introduced below.

#### 5.2.1 Encoder

**Code change embedding.** The basic ideas for embedding a code change are (1) the structure information should be taken into consideration and (2) we not only need to focus on the changed part but also need to include the unchanged part since it provides context information that can help understand which parts of the old comment should be unchanged [25], [26]. In this study, we rely on the AST path technique due to its convenience to combine code change information (through the code change operators assigned to each AST path) with program structure information (through the node sequence in each AST path). Extracting AST paths only requires the code is syntactically correct. In contrast, more advanced techniques that rely on the dependency information in code [55] usually require compilable code as input, are more heavyweight and costly, and could be explored in future, as will be discussed later in Section 7.2.

To extract the structure information, we first generate the ASTs of the old ($A_O$) and new methods ($A_N$) using a widely-adopted tool, Gumtree [56]. We then calculate the token

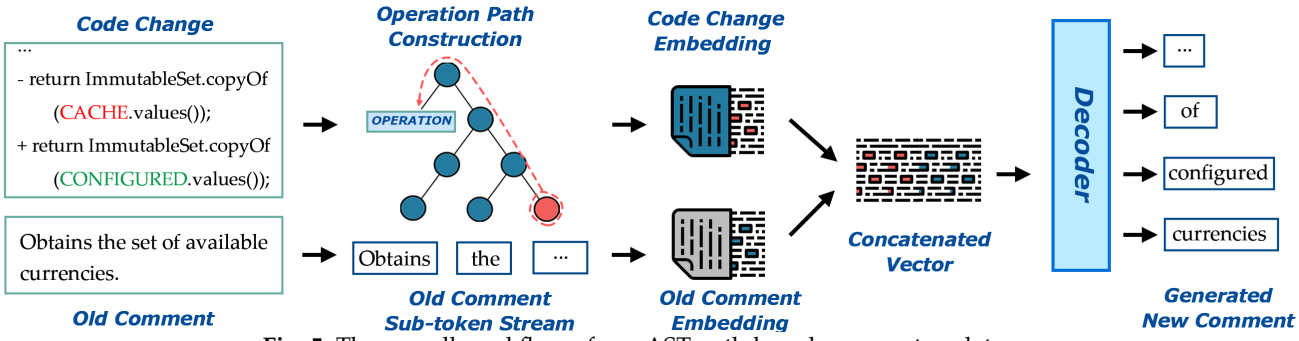



**Fig. 5:** The overall workflow of our AST-path-based comment updater.

level alignment result as $(< t_1, t'_1, o_1 >, \ldots < t_n, t'_n, o_n >)$. For each element in the sequence,

- if the operation ($o_i$) is *keep* or *delete*, we extract the AST path of $t_i$ from $A_O$ and assign the corresponding operation to construct the operation path;
- if the operation is *insert*, we extract the AST path from $A_N$ for constructing the operation path;
- if the operation ($o_i$) is *update*, we extract the AST path of $t_i$ from $A_O$ and then combine $t_i$, the path, and $t'_i$ to construct the operation path.

It should be noted here that some tokens in the alignment results cannot be matched to the leaf nodes in the AST. For example, if a code change adds a conditional block, then the added token `if` can only be matched to an interior node `IfStatement` because there is no leaf node related to `if`. We decide to ignore such nodes since their semantic meanings can be reflected by the AST path. Considering the above-mentioned example, the AST path of each added token in the conditional block must contain the node `IfStatement`. Therefore, we argue that focusing on the changes of leaf nodes is reasonable.

The AST path extraction is based on an off-the-shelf AST path collector [57] (i.e., *PathMiner*). Then, the operation paths whose change operators are *keep* ($OP_{ct} = op_1, \ldots, op_x$) are served as context while the remaining paths ($OP_{cc} = op'_1, \ldots, op'_y$) are used to represent the code change content. Consider Fig. 1 as an example. In the code change, the token `version` is changed to `updateVersion`, which results in an operation path $op'_1 = \langle version, p, updateVersion \rangle$ where $p = \langle MethodDeclaration \rightarrow Block \rightarrow ReturnStmt \rightarrow FieldAccessExpr \rangle$. Because `version` is the only modified token in the code change, the change operators assigned to other tokens in the method are all *keep*. Therefore, the $OP_{cc}$ in this case contains only one operation path (i.e., the $op'_1$).

We next introduce how to embed each operation path and how to integrate them to represent the bag of paths.

*Operation path embedding.* Given a set of $k$ operation paths $\{op_1, \ldots, op_k\}$, the model learns a vector representation $V_{op_i}$ for each path $op_i = \langle\ t_i, p_i, o_i\ \rangle$ where $p_i = \{n^i_1, n^i_2, \ldots, n^i_{l_i}\}$ is the corresponding AST path, $t_i$ is the code token and $o_i$ is the change operator. To that end, we first leverage a matrix to map each sub-token (after splitting $t_i$) into vectors and sum the sub-token vectors as the representation of the full token:

$$V_{t_i} = \sum_{st_j \in T_i} E_t(st_j)$$

where $E_t(*)$ is the learned embedding matrix, $T_i$ is the sub-token sequence of code token $t_i$. The change operator of each operation path is embedded with another learned matrix (i.e., $E_o(*)$):

$$V_{o_i} = E_o(o_i)$$

The AST path of each operation path is composed of several AST nodes. We also represent each node $n_i$ using a learned embedding matrix $E_p(*)$ and encode the entire sequence with the final state of the bi-directional LSTM neural networks:

$$\mathrm{V}_{p_i} = LSTM(E_p(n_1), E_p(n_2), \ldots, E_p(n_l))$$

where $V_{p_i}$ represents the vector representation of an AST path. $LSTM$ denotes the bi-directional LSTM neural networks. At last, the concatenation of $[V_{t_i}; V_{p_i}; V_{o_i}]$ is used to represent this operation path (i.e., $V_{op_i}$).

*Integration with attention network.* In this step, given the representations of a bag of operation paths, we pass them through a fully-connected layer and an attention network sequentially for generating the representation of the whole bag of paths (namely $V_{OP_{ct}}$). The main job of the attention mechanism is to compute a scalar weight for each path [28].

Formally, given the representations of operation paths in a bag $\{ V_{op_1}, V_{op_2}, \ldots, V_{op_k} \}$, the process can be formulated as:

$$z_i = tanh(W_f \times V_{op_i})$$
$$a_i = \frac{exp(z_i^T \cdot a)}{\sum_{j=1}^{n} exp(z_j^T \cdot a)}$$
$$V_{OP_{ct}} = \sum_{i=1}^{k} a_i \cdot z_i$$

where $W_f$ is the weight matrix of the fully-connected layer, $a$ is the attention vector which is initialized randomly and learned simultaneously with the network, $a_i$ is the attention weight of $z_i$, which is computed as the normalized inner product between $z_i$ and the global attention vector $a$, and $V_{OP_{ct}}$ is a linear combination of vectors $\{ z_1, z_2, \ldots, z_k \}$ factored by their attention weights to represent the whole bag of paths. The attention here, which determines the weights of different paths, can be generally considered as a pooling method. It is widely-used in tree-based code representation [27], [58].

Note that in the above content, we use the bag of context operation paths as an example for the ease of our presentation. The process for integrating the bag of code

change operation paths is similar which results in $V_{OP_{cc}}$. We then concatenate $V_{OP_{ct}}$ and $V_{OP_{cc}}$ as the embedding of the code change $\mathbf{V}_{CC}$.

**Old comment embedding.** Given an old comment, we parse it into a sequence of sub-tokens $(st_1, \ldots, st_x)$ and then represent it as the output of a bi-directional GRU:

$$\mathbf{V}_{CM} = GRU(E_c(st_1), E_c(st_2), \ldots, E_c(st_x))$$

where $E_c(*)$ is another learned embedding matrix for sub-tokens in comments. Here we use GRU instead of LSTM because our experiments show that using GRU to model old comments can achieve better performance.

#### 5.2.2 Decoder

As for the decoder part, we use the same model as Liu et al. [25], the *pointer generator* [59], which generates the sub-tokens of the new comment sequentially. This neural network is capable of alleviating the out-of-vocabulary (OOV) problem by copying contents from the input sequence. Another possible strategy to deal with the OOV problem would be to use Byte Pair Encoding (BPE) [60]. This study mainly focuses on helping the deep-learning-based and the heuristic-based comment updaters complement each other, so we chose to keep in line with the prior work [25], [26] and used the pointer generator. Investigating the effectiveness of BPE on JIT comment update is saved for future work.

We briefly introduce the pointer generator here and readers can refer to [25], [59] for details.

Upon generating $\mathbf{V}_{CC}$ and $\mathbf{V}_{CM}$, we concatenate them and send the resulted vector to our decoder. For each predicted sub-token, the decoder needs to decide if it is generated from a fixed vocabulary or copied from the old comment. Intuitively, if the former holds, it means one sub-token from the old comment should be changed. Therefore, the token to which this old sub-token belongs represents the location result (cf. Section 4.2.1) of this instance. To achieve its goal, the decoder first generates an j-dimensional vector via the attention mechanism.

$$\mathbf{V}_{DE} = \tanh(\mathbf{W}_{attn}[\mathbf{V}_{OP_{ct}}; \mathbf{V}_{OP_{cc}}; \mathbf{V}_{CM}])$$

The $\mathbf{V}_{DE}$ can be seen as what has been learned from the code change and old comment. We utilize it to produce the probability of generating a sub-token from the vocabulary.

$$P_{gen} = \text{softmax}(\mathbf{V}_{DE}\mathbf{W}_{gen})$$

where $\mathbf{W}_{gen}$ denotes a learnable matrix of the decoder with size $j \times K_{vocab}$. $j$ is the size of $\mathbf{V}_{DE}$, and $K_{vocab}$ is the vocabulary size. $P_{gen}$ is the probability distribution over all sub-tokens in the vocabulary.

Similarly, with the help of another matrix $\mathbf{W}_{copy}$, the decoder calculates the probability distribution of copying sub-tokens from the current old comment:

$$a^t = \text{softmax}(\mathbf{V}_{DE}\mathbf{W}_{copy})$$

where $\mathbf{W}_{copy}$ is a learnable matrix with size $j \times K_{cm}$. $K_{cm}$, the length of the old comment after padding, is empirically set to 30. Our statistic result shows that around 98% of the comments in the training set have less than 30 sub-tokens. Therefore, the decoder can capture the vast majority of semantic information of comments with $K_{cm}$ as 30. Finally, the probability of outputting the sub-token $w$ is calculated as:

$$P(w) = P_{gen}(w) + \sum_{i:w_i=w} a_i^t$$

where the first part denotes the probability of generating $w$ from the *fixed vocabulary*, while the second part denotes the probability of copying $w$ from the old comment. Note that if $w$ does not appear in the old comment, then $\sum_{i:w_i=w} a_i^t$ equals to zero.

#### 5.2.3 Loss calculation

During training, the overall loss of the predicted comment (comprised of a sub-token sequence) is calculated as the average loss at each decoding step, which is the negative log likelihood of the oracle sub-token $w_t^o$ of that step:

$$\text{loss} = \frac{1}{T}\sum_{t=0}^{T}(-\log P(w_t^o))$$

### 5.3 Rollback Strategy

Our classifier may make incorrect predictions, e.g., identifying a non-code-indicative update as code-indicative. Under such a situation, HebCup may find no matched token for the update when traversing the tokens in the old comment and thus cannot perform any update (cf. the workflow of HebCup in Section 2.2). We use a rollback strategy to cope with the inaccuracy of the classifier: if the above situation happens, we send the instance to our non-code-indicative comment updater to generate the new comment.

## 6 Evaluation

In this section, we introduce the experiment designs and results. Specifically, we aim to address the following research questions:

**RQ1:** How effective is `Toper` for JIT comment update?

**RQ2:** Can our classifier accurately determine if an instance is a code-indicative update?

**RQ3:** What is the performance of our specially designed model on non-code-indicative updates?

**RQ4:** To what extent do the two key components contribute to the overall effectiveness of `Toper`?

### 6.1 Experiment Settings

**RQ1.** Originally, our dataset is split into training, validation, and test sets. In this step, we adopted the Random Forest implementation and the default hyper-parameters provided by sklearn [61] to implement our classifier. Our classifier is trained on all the instances from the training and validation set, aiming to use more data for training.

We then trained and tuned our AST-path-based comment updater on the non-code-indicative instances from training and validation sets, respectively. For the encoder in our AST-path-based updater, we tuned the hyper-parameters with some widely-used values. Specifically, the word embedding dimension (64, 128, 192, 256) and the hidden size of GRU (128, 192, 256, 384) were tuned with all possible combinations of the listed values on the validation set. Eventually,we adopted the 128-dimensional word

embeddings for the code tokens, comment tokens, and AST nodes, after the tuning process. The two GRUs in our model are both with 256 dimensions and one layer only. For the hyper-parameters related to the AST paths, we empirically collect up to 200 paths for each method with no more than nine non-leaf nodes in each path, following the previous study [27]. Other hyper-parameters of our updater and the hyper-parameters of CUP are reused from the previous study [25]. The sizes of the vocabularies of code tokens and comment tokens are 22,635 and 26,126, respectively. Note that both vocabularies only keep the tokens appearing more than once.

After preparing the two key components of `Toper`, we evaluated the effectiveness of `Toper` on all the instances from the test set. Beyond comparing with CUP and HebCup, we also compared with the state-of-the-art comment update pipeline introduced in Section 3 to demonstrate the performance enhancement of `Toper`.

**RQ2 & RQ3.** These two RQs separately dissect the effectiveness of the key components of `Toper`. To answer them, we directly reused the trained models we have introduced above. Note that when answering RQ3, we also retrained and evaluated CUP under the same settings (i.e., only considering non-code-indicative instances from our dataset). Such a comparison can demonstrate the rationale of including the structure contexts provided by AST paths.

**RQ4.** In this RQ, we investigate the impact of the two key components in our approach, the classifier and the AST-path-based comment updater, on the effectiveness of `Toper`. To study the former, we designed such a pipeline where we send all instances to HebCup and then use the AST-path-based comment updater trained on non-code-indicative instances to update those for which HebCup does not work. To study the latter, we designed another pipeline where we first predict comment update type for each instance with the classifier and then use HebCup and CUP to deal with code-indicative and non-code-indicative updates separately.

### 6.2 Metrics

Following previous studies [24], [25], our evaluations of RQ1, RQ3, and RQ4 (those related to comment updater) focus on the metrics listed below:

- **Accuracy:** the percentage of the test samples where *correct comments* are generated at Top-1. *Correct comments* refer to those that are identical to the ground-truth (i.e., written by developers).
- **AED:** the average word-level Levenshtein distance required to change the predicted results from `Toper` into the ground-truth. This value indicates the distance between the generated comments and the ground-truth comments: the smaller, the better.
- **RED:** the average of the quotient of word-level Levenshtein distance required to change the predicted results from `Toper` into the ground-truth and word-level Levenshtein distance required to change the original comment into the ground-truth. This value indicates to what extent an approach can release developers' burden from manual updates: the smaller, the better.

Formally, the word-level Levenshtein edit distance (denoted as e_d), AED and RED are calculated as follows:

$$e\_d = \begin{cases} |a| & if |b| = 0, \\ |b| & if |a| = 0, \\ e\_d(tail(a), tail(b)) & if a[0] = b[0], \\ 1 + min \begin{cases} e\_d(tail(a), b) \\ e\_d(a, tail(b)) \\ e\_d(tail(a), tail(b)) \end{cases} & otherwise \end{cases}$$

where $|x|$ is the length of the list $x$, $tail(x)$ refers to a sub-list of $x$ that removes the first element of $x$, and $x[0]$ denotes the first element of the list $x$.

$$AED = \frac{1}{N}\sum_{n=1}^{N} e\_d(\hat{y}^{(n)}, y^{(n)})$$

$$RED = \frac{1}{N}\sum_{n=1}^{N} \frac{e\_d(\hat{y}^{(n)}, y^{(n)})}{e\_d(x^{(n)}, y^{(n)})}$$

where $N$ is the number of instances in a test set, $\hat{y}^{(n)}$ refers to the comment generated for the $n_{th}$ instance, $y^{(n)}$ refers to the comment written by developers for the $n_{th}$ instance (considered as the ground-truth), and $x^{(n)}$ refers to the comment before update of the $n_{th}$ instance.

Note that when calculating the word-level Levenshtein distance required to change the predicted results into the ground-truth, previous studies [24], [25] take into consideration the special token "<con>" used to connect subtokens of compound words, which may bring bias to the final results. In this work, we ignored this token when evaluating. Therefore, the AED and RED values of HebCup and CUP listed in this paper will be slightly different from their previously reported ones.

To answer RQ2, we measured the precision, recall, accuracy, F-score, and AUC, which are widely used in assessing predictive models [44], [51], [62], [63].

### 6.3 RQ1: Effectiveness of `Toper`

The effectiveness of `Toper` and the baseline approaches are listed in Table 8. We note that our `Toper` not only outperforms existing approaches (e.g., CUP and HebCup) but also performs better than the state-of-the-art comment update pipeline. Specifically, `Toper` achieves an accuracy that exceeds 30%, outperforming the state-of-the-art pipeline (25.8%) by around 20%. Such a value is also higher than those of the two existing approaches (HebCup and CUP), which are 25.6% and 15.8%, respectively. When it comes to AED and RED, `Toper` nearly outperforms all approaches. For instance, the RED value of `Toper` is 0.76 while those of other approaches are all higher than 0.8. This value indicates that when changing the original comment into the correct updated comment, `Toper` can help developers modify a quarter less comment tokens. The only exception happens when comparing it with HebCup in terms of AED. This is because HebCup does not perform any update when no contents from the changed code and old comment can be matched while updating this type of instance is still challenging for current techniques (we will discuss this point in detail later in Section 7.2). Being that said, `Toper`'s RED

**TABLE 8:** Performances of `Toper` and baselines on the test set.

| Approach | Accuracy | AED | RED |
|---|---|---|---|
| CUP | 15.8% | 3.02 | 0.94 |
| HebCup | 25.6% | 2.53 | 0.84 |
| HebCup + CUP | 25.8% | 2.76 | 0.88 |
| `Toper` | 30.1% | 2.55 | 0.76 |

value (which can be regarded as a normalization of AED) is still lower than that of HebCup (0.76 vs 0.84), demonstrating the effectiveness of our approach. We also conducted Wilcoxon signed-rank tests [64] where we compared the achieved AED and RED values of `Toper` on each individual instance against those achieved by the baselines, and results show that the improvements achieved by `Toper` are all significant except for comparing against HebCup on AED. Specifically, when comparing the AED values of `Toper` with those of HebCup, the p-value is 0.03; while the p-values are all less than 0.001 among other comparison results.

*Our `Toper` improves the state-of-the-art by around 20% concerning the number of generated correct comments. The RED value of `Toper` is also systematically better than those of the existing approaches.*

### 6.4 RQ2: Performance of the Classifier

The performance of the classifier is presented in Table 9. According to the correlation analysis and redundancy analysis presented in Section 4.1, there are no strong correlations among the features. Thus, to dissect the contribution of each included feature, we also performed an ablation study where we removed one feature at a time and re-trained our classifier based on the rest features and re-assessed its effectiveness. Please note that in this RQ, our main aim is to show that every feature utilized by our classifier is reasonable and beneficial. Consequently, we dissect the contribution from each individual feature although there may have weak associations between features(as we have analyzed in Section 4.1).

From the results, our classifier generally works well, with an accuracy exceeding 80% and an F-score of 86%. We also note that each included feature contributes to the overall performance, more or less. The feature NSOD contributes more significantly than others, without which the accuracy will drop from 82.3% to 73.6% (a decrease of 10.6%), the F-score will drop from 86.0% to 78.1% (a decrease of 9.2%), and the AUC will drop from 91.8% to 84% (a decrease of 8.5%). This is consistent with our Cliff's delta result which shows that this feature has large effect size. The most subtle contribution is from the statement type context information, without which the accuracy, F-score, and AUC of our classifier will only decline 0.7%, 0.8%, and 0.7%, respectively. Besides, we note that all the four features (NMT, NML, NMC, and NNTPR) whose effect sizes are identified as negligible through the Cliff's delta results make non-negligible contributions. For instance, without NNTPR, the F-score of the classifier will drop from 86.0% to 84.2%, a decrease of 1.8%. The experimental results indicate that the features selected through our empirical analysis are all reasonable. Furthermore, we performed another experiment where the feature TE is involved. Specifically, we considered the TE feature for the first three changed tokens (such a number is chosen to keep in consistent with the feature TS) which led to the feature vector being 14-dimensional. Note that if the number of changed tokens is less than three, we will pad it with the default expression type. We retrained the Random Forest model and reevaluated it. Results are also presented in Table 9, which show that the values of all the metrics decrease slightly. For instance, the F-score decreases from 86.0 to 84.3. Such results illustrate that excluding the TE feature from our classifier, which is inspired by our empirical study, is rational.

**TABLE 9:** Performances of the variants of our classifier on differentiating two types of updates (in %).

| Features | Precision | Recall | Accuracy | F-score | AUC |
|---|---|---|---|---|---|
| -NSOD | 89.4 | 69.3 | 73.6 | 78.1 | 84.0 |
| -NMS | 93.7 | 76.3 | 80.5 | 84.1 | 89.9 |
| -NNTPR | 93.3 | 76.6 | 80.5 | 84.2 | 90.2 |
| -NTOD | 93.1 | 76.9 | 80.5 | 84.2 | 90.5 |
| -NMT | 92.5 | 77.5 | 80.5 | 84.4 | 90.3 |
| -NNSPR | 93.7 | 77.4 | 81.2 | 84.8 | 90.6 |
| -NML | 93.7 | 77.5 | 81.3 | 84.9 | 91.0 |
| -NMC | 93.8 | 77.6 | 81.4 | 85.0 | 91.1 |
| -TS | 93.6 | 78.1 | 81.6 | 85.2 | 91.2 |
| +TE | 94.3 | 76.2 | 80.8 | 84.3 | 90.9 |
| Our classifier (9 features) | **94.7** | **78.7** | **82.3** | **86.0** | **91.8** |

*Our classifier can generally work well and each considered feature contributes positively to its overall effectiveness with NSOD being the most rewarding one.*

### 6.5 RQ3: Performance of the AST-Path-Based Comment Updater

From the evaluation results shown in Table 10, we note that our AST-path-based comment updater significantly outperforms CUP on non-code-indicative updates. Specifically, the accuracy of our approach is 7.40% while that of CUP is only 2.70%, meaning that the number of correct updates generated by our approach is nearly twice larger than that of CUP. From the perspective of AED and RED, comments generated by our updater are closer to the ground-truth with respect to the word-level Levenshtein distance and thus save more human efforts for the developers, compared against those produced by CUP. For instance, to transform the prediction results of CUP into the corresponding ground-truth, we need to modify nearly four tokens on average, while the number for our updater is less than 3.7. Also, our statistical test results show that the p-values of our AST-path-based updater compared with CUP in terms of the three metrics are all less than 0.001.

Beyond whether the correct comments are generated, we also investigated the performances of our updater and CUP on locating the comment tokens that need to be updated. Among the 8,201 tokens within the non-code-indicative updates that are updated by developers, our comment updater modifies 2,146 of them while CUP only modifies 813 of them. Such a proportion is consistent with the accuracy values of the two approaches. Thus, the possible reason that our approach generates more correct comments than CUP is that our approach successfully predicts more tokens that should be updated. The results also show that compared with considering the code as the token sequence, incorporating the structure information when embedding code change

**TABLE 10:** Performances of our AST-path-based updater and CUP on non-code-indicative updates.

| Approach | Accuracy | AED | RED |
|---|---|---|---|
| CUP | 2.70% | 3.99 | 1.16 |
| AST-path-based updater | 7.40% | 3.68 | 0.99 |

**TABLE 11:** Performances of variants of `Toper`.

| Approach | Accuracy | AED | RED |
|---|---|---|---|
| HebCup + AST-path-based updater | 28.5% | 2.68 | 0.86 |
| HebCup + CUP + Classifier | 27.4% | 2.77 | 0.85 |
| HebCup + Original + Classifier | 24.2% | 2.70 | 0.76 |
| `Toper` | 30.1% | 2.55 | 0.76 |

does help locate the tokens to be changed, which illustrates the rationality of our motivation.

*Our AST-path-based comment updater can generate nearly 3x as many correct updates as CUP on non-code-indicative updates, which is probably because our approach can predict the comment tokens to be updated more precisely.*

### 6.6 RQ4: Ablation Study

Results shown in Table 11 reveal that both the classifier and the AST-path-based comment updater are irreplaceable component of our approach. Specifically, if we do not use the classifier to predict instances fed to Hebcup, the accuracy will drop from 30.1% to 28.5%, a decrease of 5.3%; if we do not use the specially designed updater for non-code-indicative ones but directly reuse CUP, the accuracy will decline even more significantly, from 30.1% to 27.4% with a decrease of 9.0%. Similarly, the AED and RED values of the two variants are both higher than those of `Toper` respectively, indicating that developers may save more time on comment update based on the results from `Toper`. Moreover, we conducted Wilcoxon signed-rank tests [64] where we compared the achieved AED and RED values of `Toper` on each individual instance against those achieved by the two variants. Results reveal the p-values in terms of the three metrics are all less than 0.001, which means the performance improvements achieved by `Toper` over the two variants are statistically significant.

From the results in Table 10, the RED value of our AST-path-based updater is around 1, indicating that once applied, developers may still need to modify the updated comments substantially. Therefore, we also investigate the effectiveness of another pipeline where we perform no update for those predicted as non-code-indicative by our classifier, to show the difference between using or not using the AST-path-based updater. Results are also shown in Table 11. We note that although the RED value of this pipeline is similar to that of `Toper` (both of which are around 0.76), its accuracy value is much lower (24.2 vs. 30.1%), which means that excluding our AST-path-based updater will miss the opportunity of generating a certain number of oracle comments directly.

*Both the classifier and the AST-path-based comment updater contribute significantly to the performance of `Toper`, without which the accuracy of our approach will decrease $\approx$ 5% and $\approx$ 9% respectively.*

## 7 Discussion

### 7.1 Can the AST-Path-Based Comment Updater Play Alone?

With the help of program structure information, our AST-path-based comment updater significantly outperforms CUP on non-code-indicative (cf. Table 10). A further question deserves exploration is that how effective the AST-path-based updater is on all the update instances (including both code-indicative and non-code-indicative ones). To investigate this, we retrained and evaluated our AST-path-based comment updater on all the instances in our dataset. We recall here that in the previous performed experiments, this updater was trained merely on the non-code-indicative updates.

Our experiment results show that the AST-path-based updater achieves an accuracy value of 16.0%. This effectiveness is similar to that of CUP which is 15.8%. Such results indicate that our AST-path-based updater performs poorly on code-indicative updates. This is reasonable considering that our motivation for designing such an updater is only gained through focusing on CUP's weakness on non-code-indicative updates. This, however, may shed light on further exploration that combining the textual and structural information of the code is a possible way. Overall, our results show that it is hard currently to propose a single approach that is effective for both code-indicative and non-code-indicative updates. The rationale of our approach to adopt different ways for different types of updates is thus further demonstrated.

### 7.2 Future Directions for Comment Update

Despite that our `Toper` significantly improves the state-of-the-art, we note that there is still a large space for improvements. One aspect is how to deal with non-code-indicative updates more effectively. Specifically, although our AST-path-based updater outperforms the existing approach (i.e., CUP) a lot on this type of update, its RED value is still around 1 (cf. Table 10). Generally speaking, the value indicates that the average word-level Levenshtein distance between the updated comment and the oracle new comment is not significantly lower than that between the old comment and the oracle new comment, suggesting that once being applied, developers may not expect to spend less efforts in updating the comments. We give a concrete example in Table 12. In this case, developers add a parameter into a method invocation. By investigating the called function, we found that it usually parses an object. However, if the input is set to -2, it will perform some special operations towards the format of the object. Therefore, to illustrate the effect of this parameter, developers change the verb in the comment from "parse" to "format". `Toper`, however, only captures the added textural content -2 without understanding the functionality of the called method. It thus generates a wrong update as shown in the table. Under such a situation, the word-level Levenshtein distance between the old and new comments is one while that between the new and `Toper` generated comments is 4, resulting in the RED value become relatively high (i.e., 4). Therefore, it calls for more advanced techniques to capture the intention of code changes and better infer the non-code-indicative updates in future studies.

**TABLE 12:** A non-code-indicative instance incorrectly updated by `Toper`.

| |
|---|
| **Code Change**:<br>public DateTimeFormatterBuilder appendInstant() {<br>− appendInternal(new InstantPrinterParser());<br>+ appendInternal(new InstantPrinterParser(-2));<br>return this;<br>} |
| **Old Comment**: Parse the instant as a single epoch-seconds value. |
| **New Comment**: Format the instant as a single epoch-seconds value. |
| **`Toper`**: Parse the instant as a -2. |

**TABLE 13:** A non-code-indicative instance misclassified by the classifier.

| |
|---|
| **Code Change**:<br>− public ImmutableMap<PositionAttributeType<?>, Object> getAttributes(){<br>+ public ImmutableMap¡AttributeType<?>, Object> getAttributes() {<br>return attributes;<br>} |
| **Old Comment**: Attributes provide the ability to associate arbitrary information with a position in a key-value map. |
| **New Comment**: Attributes provide the ability to associate arbitrary information ~~with a position~~ in a key-value map. |
| **HebCup**: Attributes provide the ability to associate arbitrary information with a ~~position~~ in a key-value map. |

For this example, incorporating the caller/callee information may help boost the effectiveness. Beyond that, some more advanced static-analysis-based code embedding approaches (e.g., using data-flows and control-flows) [55], [65] would be beneficial to capture the code change semantics and thus deserve exploration in future studies.

Another aspect that needs to be enhanced is the accuracy of the classifier. Specifically, although the overall accuracy of the classifier exceeds 80%, there still exists cases where its inaccuracy may cause side effects. A concrete example is shown in Table 13. According to the code change, the type of the key in the returned map is changed from `PositionAttributeType` to `AttributeType`, which means the specific position information no longer exists in the returned map. Therefore, developers removed "with a position" in the old comment to reflect this semantic change. In this code change, the disappeared sub-token (i.e., position) exists in the old comment, making the value of the feature NSOD be 1. From Fig. 7, when the value of this feature is larger than 1, it is more likely that it is a code-indicative update. Indeed, our classifier mistakenly identifies this case as a code-indicative one and thus `Toper` uses HebCup to generate the incorrect comment, in which only `position` in the old comment is removed since it matches the code change content. We have checked the results from RQ3 and confirmed that our AST-path-based updater can generate the correct new comment for this instance. This suggests in the future by building a more accurate classifier, we may further boost the overall performance of `Toper`. We also assess the performance of another pipeline where we assume the classifier can achieve the 100% precision in differentiating the code-indicative and non-code-indicative updates. Results show that such a pipeline can achieve an accuracy of 34.5%, an AED of 2.52, and a RED of 0.747, respectively, which indicates that, the enhancement of the classifier's accuracy could lead to the performance improvement of `Toper`.

Although HebCup generally achieves promising performance on code-indicative updates (it correctly updates

**TABLE 14:** A code-indicative instance updated incorrectly by HebCup.

| |
|---|
| **Code Change**:<br>− public void deleteQuery(Query query) {<br>+ public void deleteEntry(PlaylistEntry entry) {<br>... } |
| **Old Comment**: Remove query at given position from current playlist. |
| **New Comment**: Remove entry at given position from current playlist. |
| **HebCup**: Remove PlaylistEntry at given position from current playlist. |

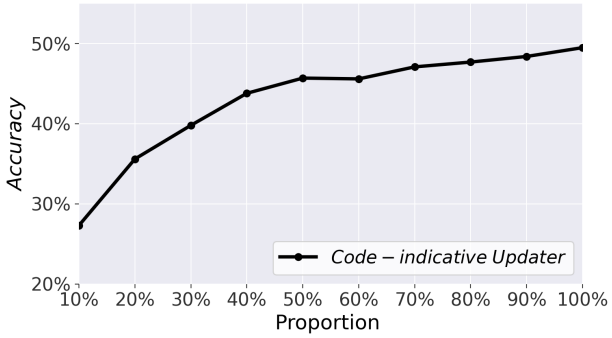



**Fig. 6:** The performance of code-indicative updater for different proportions of code-indicative instances from the training set.

2,355/2,970 of the code-indicative ones with the accuracy being nearly 80%), the heuristic-based approach has its own inherent limitation (i.e., the inaccuracy brought by the fixed heuristic). One example is shown in Table 14 where the comment token "query" is changed into "PlaylistEntry" but should be "entry". In this example, two replacement pairs (i.e., "query" → "entry" and "query" → "PlaylistEntry") are established according to the code change. From the heuristic of HebCup, if one token has multiple candidate tokens for replacement, the one with the largest number of sub-tokens will be prioritized. Therefore, "PlaylistEntry" is selected for update under this situation, resulting in the incorrect update. To overcome this limitation, one direction in the future would be training better deep learning models with more code-indicative data. Such learning-based models would be more flexible than heuristics, and thus hold the potential to boost the effectiveness of comment update. Although existing learning-based approaches have the limitations [24], i.e., (1) they frequently ignore some code change information and (2) they are often misled by noisy information in the complicated code change, to explore the feasibility of this future direction, we performed another experiment where we trained an AST-path-based code-indicative updater with different percentages of code-indicative updates from the training set (10% to 100% with an interval of 10%) and then evaluated it on the code-indicative updates from the test set. Results shown in Fig. 6 reveal that the effectiveness of such an updater consistently increases with more training data, illustrating that enlarging the training data can improve the performance of deep learning-based models on the code-indicative updates.

Another potential way to improve the effectiveness of the heuristic-based approach is to summarize and apply more heuristics. For example, from the instance shown in Fig. 3a, we foresee a way to facilitate the comment update, which is expanding the abbreviations in the code [66], [67]. In this example, the token `millis` in the code is an abbreviation of the token `milliseconds` in the comment. Therefore, if we can map these two tokens, this instance can

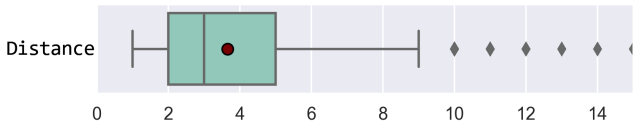



**Fig. 7:** The distribution of the Levenshtein distance required to change each incorrect update into its corresponding ground-truth.

be considered as a code-indicative one, which is easier to update.

Accordingly, the main root causes for incorrect updates are (1) the ineffectiveness of current models in capturing the semantic information of complicated code changes, (2) the ineffectiveness of current approaches in building code-comment traceability, and (3) the update type misclassification caused by the classifier. To conclude, our analysis reveals four potential ways to improve comment update effectiveness, which are (1) a better model to capture code change semantics, (2) a better classifier to predict if an instance is code-indicative or not, (3) more effective model with more training data, and (4) more heuristics to establish the semantic interaction between the code and comment, hold the potential to obtain a higher accuracy of comment update. It is quite hard to say which improvement way would be more effective than others so that all of them could be explored in the future. However, comprehensively investigating which improvement is more effective than others is non-trivial hard and out of this work's scope. So we leave this for future work.

### 7.3 The Characteristics of the Updated Comments

To provide more insights for future studies, we analyze the updated comments from another two perspectives in this section. First, we investigate the word-level Levenshtein distance required to change each incorrect update into its corresponding ground-truth. As shown in Fig. 7, the medium value is 3 and the upper quarter is 5. Such results indicate that for most conditions, changing the updates generated by `Toper` into the ground-truth ones requires the modification of no more than 5 tokens. This suggests the opportunity of generating more correct updates for future studies.

Second, we aim to investigate if the number of updated comment tokens that do not appear in the code change will affect the effectiveness of our AST-path-based updater. To this end, we design a finer-grained categorization of non-code-indicative updates based on the number of comment tokens that cannot be borrowed from the code change directly ($NUM_{cbb}$). The effectiveness of AST-path-based updater on each type is shown in Table 15. Results reveal that with the increase of $NUM_{cbb}$, the performance of our updater will generally drop down (i.e., the accuracy decreases and the AED increases). Recall that the accuracy value of HebCup on code-indicative updates reaches around 80% (mentioned in the last section), such results call for improvements to capture the semantics of the comment updates that are implicitly reflected by the code changes, especially for those updates that involve multiple tokens which cannot be reused from the code changes.

**TABLE 15:** The performance of the AST-path-based updater on each finer-grained type of non-code-indicative updates.

| Category | Proportion | Accuracy | AED | RED |
|---|---|---|---|---|
| 1 | 42.1% | 13.1% | 3.09 | 1.01 |
| 2 | 17.0% | 3.1% | 3.29 | 1.00 |
| 3 | 12.3% | 3.8% | 3.62 | 0.99 |
| 4 | 9.7% | 3.1% | 4.02 | 0.99 |
| $\geq 5$ | 18.9% | 3.5% | 5.20 | 0.97 |
| All | 100% | 7.4% | 3.68 | 0.99 |

"Category" denotes the number of updated comment tokens that cannot be borrowed directly from code changes.

### 7.4 The Replication Study of Another Existing Deep Learning based Approach

We note that Panthaplackel et al. also proposed a learning based updater [26]. However, this approach was only evaluated on comments starting with `@return`. To assess its generalizability to other comments, we re-trained and re-evaluated it on the dataset in this study based on the open-sourced replication package provided by the authors. Table 16 shows the evaluation results. This approach achieved similar performances with those of CUP on all the instances in the test set. Specifically, its accuracy value is identical to that of CUP (i.e., 15.8%), and its AED value is very similar to that of CUP (3.01 vs. 3.02). When it comes to the non-code-indicative updates in the test set, we also observe the same phenomenon. Indeed, the working mechanisms of these two approaches is nearly identical: both approaches utilize the textual information from the code and comment, and train learning-based models to generate the updated comments. Therefore, in this study, we use CUP as a representative tool of the existing deep-learning-based techniques. We design our approach based on our investigation towards the weakness of CUP and the results show that our approach can outperform the existing approaches significantly on both non-code-indicative updates and all the instances in the test set, as shown in Table 16.

### 7.5 The Generalization of Our Approach

In this study, we dissect the weaknesses of current approaches based on their results on the test set, after which we design our own approach that can outperform them on this set. There is thus a concern that to what extent can our approach generalize to instances that are not involved in the test set. Following the previous study [24], we performed another experiment to address this concern, where we re-trained CUP and our AST-path-based updater by keeping the training set unchanged, using the test set for validation and testing the models on the validation set. The classifier was accordingly trained on all the instances from the training and the test sets. Results are shown in Table 17. We note that (1) `Toper` achieves very similar performance compared with that listed in Table 8; and (2) `Toper` outperforms all the baseline approaches on all the metrics. Specifically, `Toper` still generates the oracle updates for around 30% instances in the validation set, and its AED score (2.50) is even lower than that of HebCup (2.54). Such results indicate that our approach can generalize well to new cases excluded from the test set where we discover the weaknesses of the existing approaches and design the strategy of our own approach.

**TABLE 16:** Performances of `Toper` and two existing deep learning based approach on the test set.

| Update type | Approach | Accuracy | AED | RED |
|---|---|---|---|---|
| Non-code-indicative | CUP | 2.7% | 3.99 | 1.16 |
| | Panthaplackel et al. | 3.1% | 3.83 | 1.16 |
| | AST-path-based updater | 7.4% | 3.68 | 0.99 |
| All instances | CUP | 15.8% | 3.02 | 0.94 |
| | Panthaplackel et al. | 15.8% | 3.01 | 1.01 |
| | Toper | 30.1% | 2.55 | 0.76 |

**TABLE 17:** Performances of `Toper` and baselines on the validation set.

| Approach | Accuracy | AED | RED |
|---|---|---|---|
| CUP | 19.3% | 2.99 | 0.93 |
| HebCup | 25.5% | 2.54 | 0.84 |
| HebCup + CUP | 25.6% | 2.71 | 0.89 |
| `Toper` | 29.7% | 2.50 | 0.78 |

### 7.6 Threats to Validity

**External validity.** One threat to external validity is that we only target Java projects and Javadoc comments in this study. Therefore, the effectiveness of our approach on projects of other programming languages and other types of comments (e.g., inline comments) remains unknown. This threat is mitigated considering that (1) Java is one of the most popular programming languages and (2) Javadoc comment is a type of critical information for developers to comprehend the code [68]. Moreover, our `Toper` is independent of programming languages and comment types. It can be adopted to other languages where the AST can be generated (e.g., C#) as well as to other types of comments once being reasonably trained.

Another threat is from the representativeness of our dataset. All our empirical findings that support the design of `Toper` and our evaluations are solely based on this dataset. However, our dataset contains method-comment co-change instances from up to 1,500 popular GitHub repositories. It is also carefully curated by existing studies [24], [25]. Therefore, we believe the threat is limited.

**Internal validity.** To perform the various comparison experiments in this study, we need to re-run the existing approaches (i.e., CUP and HebCup) for many times (e.g., to answer RQ3, we re-run CUP on non-code-indicative updates). Any different minor setting in this process may lead to major differences [69]. To mitigate this, we directly reuse the replication packages provided by the existing studies [24], [25] and keep the values of all the parameters the same as the original studies.

## 8 Related Work

This work targets the JIT comment update task. Literature approaches that are tightly related to our study have been detailed in Section 2.2. In this section, we revisit works in the literature that are related to code comment, deep learning for program comprehension, and AST path technique.

### 8.1 Automated Comment Generation

One way to help developers reduce the burden of program comprehension is to generate the comment for a piece of code automatically. Such a topic has been studied for more than one decade in the research community. Similar to comment update techniques, the literature approaches for comment generation can also be mainly split into two categories: heuristic based and deep learning based. Sridhara et al. [70] proposed to generate natural language summarization for methods via manually defined templates. ColCom [71] generates a comment for a code snippet by reusing comments of open source code snippets that are similar to the studied one. Panichella et al. [72] generated the summarization of test cases by first obtaining the coverage information and then exploiting Java naming conventions to construct readable natural language sentences. Hu et al. [73] and LeClair et al. [32] designed deep learning models to generate comments by integrating source code with structure information. It should be noted that the comment generation task is orthogonal to the aim of this paper since we aim to update comments that are already available with code changes. In contrast, comment generation focuses on generating comments from scratch.

### 8.2 Inconsistent Comment Detection

Given that the inconsistency between the code and its associating comment may prevent developers from clearly understanding the program, it is of great importance to detect this kind of inconsistency. Tan et al. [68] tried to infer program properties from the Javadoc comments and then randomly generated tests to check the inferred properties. Ratol and Robillard [16] proposed to detect comments that become inconsistent during identifier renaming via a set of human-written rules. Zhou et al. [74] focused on defects of Application Programming Interface (API) documentation and detected such issues based on the analysis results from a constraint solver. Liu et al. [75] identified the outdated comments via a machine learning classifier trained by mining the historical data of software repositories. Wen et al. [17] performed a large-scale investigation towards the introduction/fixing of code-comment inconsistencies where the empirical results can better guide the design of novel approaches towards the detection of inconsistent comments. However, even if the inconsistency is detected, developers still need to manually update the comments, which is rather time-consuming. Our approach, which aims at timely updating the corresponding comment given the code change, can help developers further alleviate the inconsistency between the code and the comment.

### 8.3 Deep Learning in Program Comprehension

Deep learning techniques have also been widely adopted in a number of program comprehension tasks beyond comment update. Nguyen et al. [76] and Wang et al. [29] suggested high-quality names for methods following a seq2seq paradigm, while Alon et al. [27], [28] finished the same task by exploiting the code structure information. He et al. [77] used a dual-channel Convolutional Neural Networks (CNN) model to represent a bug report pair that can effectively detect duplicate bug reports. Liu et al. [78] captured both the structure information and long-term dependency relations of an input program through a self-attentional neural architecture for code completion task. To automatically generate pull request descriptions, Liu et al. [79] designed a novel seq2seq model where the pointer generator and reinforcement learning are integrated. Our

study differs from the abovementioned ones with respect to two perspectives. First, our task (i.e., JIT comment update) is different from the mentioned studies since the inputs of our approach are the code change plus with the old comment, and the output is the updated comment, while none of the works mentioned above model code changes and comments at the same time. So they can not be used to tackle our task. Second, our approach combines deep learning techniques with heuristics through a predictive model while the listed studies rely merely on deep learning models.

### 8.4 AST Path Technique

Alon et al. [80] first pointed out that using paths in the program's AST significantly lowers the learning effort (compared to learning over program text) and is still scalable and general. This approach can capture both program syntactic (via the connection relation between two adjacent AST nodes) and semantic (via tokens of the leaf nodes) information well. After that, this code representation technique has been applied to a number of software engineering tasks including method name recommendation [28], code summarization [27], and code completion [58]. In our study, we choose to utilize this code representation technique after identifying that the structural context of the code change may boost the update of non-code-indicative instances.

## 9 Conclusion

We introduce `Toper` in this study, a two-phase approach for automated comment update. The core idea is to effectively combine different ways to deal with comment updates whose contents can or cannot be reflected by the corresponding code changes (i.e., code-indicative and non-code-indicative comment updates), respectively. Through our pre-study experiments, we identified two main challenges for such a pipeline: 1) how to precisely predict the types of the comment updates; and 2) how to effectively update non-code-indicative ones. To tackle such challenges, we performed empirical studies through which we identified nine discriminative features between code-indicative and non-code-indicative updates and found that program structure information can help deal with non-code-indicative updates. Therefore, `Toper` first utilizes a classifier which takes the nine identified features into consideration to predict the types of the comment updates. If it is a code-indicative one, the off-the-shelf approach HebCup is adopted for generating the new comment; otherwise, the AST-path-based comment updater is trained and used with the structure information of code change considered. Our extensive experiments show that `Toper` significantly outperforms the state-of-the-art and all our design decisions make sense.

**Artefacts:** All data in this study are publicly available at:

**https://zenodo.org/record/5340569**.

## Acknowledgments

This work is supported by the National Natural Science Foundation of China No.61672529.

## References

[1] Y. Padioleau, T. Lin, 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, p. 331–341.

[2] L. Pascarella, M. Bruntink, and A. Bacchelli, "Classifying code comments in java software systems," *Empirical Software Engineering*, vol. 24, pp. 1499–1537, 2019.

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

[4] J. Zhang, S. Panthaplackel, P. Nie, R. J. Mooney, J. J. Li, and M. Gligoric, "Learning to generate code comments from class hierarchies," *arXiv preprint arXiv:2103.13426*, 2021.

[5] E. Shi, Y. Wang, L. Du, J. Chen, S. Han, H. Zhang, D. Zhang, and H. Sun, "Neural code summarization: How far are we?" *arXiv preprint arXiv:2107.07112*, 2021.

[6] A. Ciurumelea, S. Proksch, and H. C. Gall, "Suggesting comment completions for python using neural language models," in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2020, pp. 456–467.

[7] H. Hata, C. Treude, R. G. Kula, and T. Ishio, "9.6 million links in source code comments: Purpose, evolution, and decay," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 1211–1221.

[8] S. Woodfield, H. Dunsmore, and V. Shen, "The effect of modularization and comments on program comprehension," in *International Conference on Software Engineering (ICSE)*, 1981.

[9] 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 IEEE Working Conference of Mining Software Repositories*, 2005.

[10] M.-A. Storey, J. Ryall, R. Ian Bull, D. Myers, and J. Singer, "Todo or to bug: Exploring how task annotations play a role in the work practices of software developers," in *2008 ACM/IEEE 30th International Conference on Software Engineering*, 2008, pp. 251–260.

[11] A. Mastropaolo, E. Aghajani, L. Pascarella, and G. Bavota, "An empirical study on code comment completion," *arXiv preprint arXiv:2107.10544*, 2021.

[12] J. Zhang, S. Panthaplackel, P. Nie, J. Li, R. Mooney, and M. Gligoric, "Leveraging class hierarchy for code comprehension," in *NeurIPS 2020 Workshop on Computer-Assisted Programming*, 2020. [Online]. Available: https://openreview.net/forum?id=WHjjpMNZFoD

[13] P. Rani, S. Abuka, N. Stulova, A. Bergel, and O. Nierstrasz, "Do comments follow commenting conventions? a case study in java and python," *arXiv preprint arXiv:2108.10766*, 2021.

[14] D. Gros, H. Sezhiyan, P. Devanbu, and Z. Yu, "Code to comment "translation": Data, metrics, baselining & evaluation," in *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2020, pp. 746–757.

[15] L. Tan, D. Yuan, G. Krishna, and Y. Zhou, "/*icomment: Bugs or bad comments?*/," in *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*, 2007, p. 145–158.

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

[17] F. Wen, C. Nagy, G. Bavota, and M. Lanza, "A large-scale empirical study on code-comment inconsistencies," in *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, 2019, pp. 53–64.

[18] S. Panthaplackel, J. J. Li, M. Gligoric, and R. Mooney, "Deep just-in-time inconsistency detection between comments and source code," in *AAAI*, 2021.

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

[20] 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, pp. 2293–2304, 2012.

[21] H. Siy and L. G. Votta, "Does the modern code inspection have value?" in *Proceedings IEEE International Conference on Software Maintenance (ICSM)*, 2001, pp. 281–289.

[22] Z. Gao, X. Xia, D. Lo, J. Grundy, and T. Zimmermann, "Automating the removal of obsolete todo comments." New

York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3468264.3468553

[23] C. Treude, J. Middleton, and T. Atapattu, "Beyond accuracy: Assessing software documentation quality," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 1509–1512.

[24] B. Lin, S. Wang, K. Liu, X. Mao, and T. F. Bissyandé, "Automated comment update: How far are we?" in *2021 29th International Conference on Program Comprehension (ICPC)*, 2021.

[25] Z. Liu, X. Xia, M. Yan, and S. Li, "Automating just-in-time comment updating," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2020.

[26] S. Panthaplackel, P. Nie, M. Gligoric, J. J. Li, and R. Mooney, "Learning to update natural language comments based on code changes," in *ACL*, 2020.

[27] U. Alon, S. Brody, O. Levy, and E. Yahav, "code2seq: Generating sequences from structured representations of code," in *Proceedings of the 7th International Conference on Learning Representations*. OpenReview.net, 2019.

[28] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: learning distributed representations of code," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 40:1–40:29, 2019.

[29] S. Wang, M. Wen, B. Lin, and X. Mao, "Lightweight global and local contexts guided method name recommendation with prior knowledge," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021.

[30] S. Wang, K. Liu, B. Lin, L. Li, J. Klein, X. Mao, and T. F. Bissyandé, "Beep: Fine-grained fix localization by learning to predict buggy code elements," 2021.

[31] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation," in *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*, 2018.

[32] 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.

[33] Y. Zhou, C. Wang, X. Yan, T. Chen, S. Panichella, and H. C. Gall, "Automatic detection and repair recommendation of directive defects in java api documentation," *IEEE Transactions on Software Engineering*, vol. 46, pp. 1004–1023, 2020.

[34] S. Wang, M. Wen, X. Mao, and D. Yang, "Attention please: Consider mockito when evaluating newly proposed automated program repair techniques," in *Proceedings of the 23rd Evaluation and Assessment on Software Engineering*. ACM, 2019, pp. 260–266.

[35] S. Wang, M. Wen, L. Chen, X. Yi, and X. Mao, "How different is it between machine-generated and developer-provided patches? an empirical study on the correct patches generated by automated program repair techniques," in *Proceedings of the 13th International Symposium on Empirical Software Engineering and Measurement*. IEEE, 2019, pp. 1–12.

[36] M. Motwani, S. Sankaranarayanan, R. Just, and Y. Brun, "Do automated program repair techniques repair hard and important bugs?" *Empirical Software Engineering*, vol. 23, no. 5, pp. 2901–2947, 2018.

[37] S. Kim, E. J. Whitehead, and Y. Zhang, "Classifying software changes: Clean or buggy?" *IEEE Transactions on Software Engineering*, vol. 34, no. 2, pp. 181–196, 2008.

[38] Q. Xin and S. P. Reiss, "Leveraging syntax-related code for automated program repair," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, 2017, pp. 660–670.

[39] X.-B. D. Le, D.-H. Chu, D. Lo, C. Le Goues, and W. Visser, "S3: syntax-and semantic-guided repair synthesis via programming by examples," in *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering*. ACM, 2017, pp. 593–604.

[40] V. Sobreira, T. Durieux, F. Madeiral, M. Monperrus, and M. de Almeida Maia, "Dissection of a bug dataset: Anatomy of 395 patches from defects4j," in *Proceedings of the 25th International Conference on Software Analysis, Evolution and Reengineering*. IEEE, 2018, pp. 130–140.

[41] S. Panthaplackel, M. Gligoric, R. Mooney, and J. J. Li, "Associating natural language comment and source code entities," in *AAAI*, 2020.

[42] H. B. Mann and D. R. Whitney, "On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other," *The Annals of Mathematical Statistics*, vol. 18, no. 1, pp. 50–60, 1947.

[43] N. Cliff, "Ordinal methods for behavioral data analysis," 1996.

[44] Y. Fan, X. Xia, D. Lo, and A. E. Hassan, "Chaff from the wheat: Characterizing and determining valid bug reports," *IEEE transactions on software engineering*, vol. 46, no. 5, pp. 495–525, 2018.

[45] A. Mockus, M. Nagappan, and A. Hassan, "Best practices and pitfalls for statistical analysis of se data," in *Proc. ICSE*, 2014.

[46] Y. Tian, M. Nagappan, D. Lo, and A. E. Hassan, "What are the characteristics of high-rated apps? a case study on free android applications," in *2015 IEEE international conference on software maintenance and evolution (ICSME)*. IEEE, 2015, pp. 301–310.

[47] W. Ziarko and S. Ning, "Machine learning through data classification and reduction," *Fundamenta Informaticae*, vol. 30, no. 3, pp. 373–382, 1997.

[48] T. R. Patil and S. S. Sherekar, "Performance analysis of naive bayes and j48 classification algorithm for data classification," *International journal of computer science and applications*, vol. 6, no. 2, pp. 256–261, 2013.

[49] D. G. Kleinbaum, K. Dietz, M. Gail, M. Klein, and M. Klein, *Logistic regression*. Springer, 2002.

[50] A. Arcuri and L. Briand, "A practical guide for using statistical tests to assess randomized algorithms in software engineering," in *2011 33rd International Conference on Software Engineering (ICSE)*. IEEE, 2011, pp. 1–10.

[51] S. Wang, M. Wen, B. Lin, H. Wu, Y. Qin, D. Zou, X. Mao, and H. Jin, "Automated patch correctness assessment: How far are we?" in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2020.

[52] L. Bao, X. Xia, D. Lo, and G. C. Murphy, "A large scale study of long-time contributor prediction for github projects," *IEEE Transactions on Software Engineering*, 2019.

[53] B. Ghotra, S. McIntosh, and A. E. Hassan, "Revisiting the impact of classification techniques on the performance of defect prediction models," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 789–800.

[54] J. Zhang, L. Zhang, M. Harman, D. Hao, Y. Jia, and L. Zhang, "Predictive mutation testing," *IEEE Transactions on Software Engineering*, vol. 45, no. 9, pp. 898–918, 2018.

[55] Y. Sui, X. Cheng, G. Zhang, and H. Wang, "Flow2vec: value-flow-based precise code embedding," *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, pp. 1–27, 2020.

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

[57] V. Kovalenko, E. Bogomolov, T. Bryksin, and A. Bacchelli, "Pathminer: a library for mining of path-based representations of code," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 2019, pp. 13–17.

[58] U. Alon, R. Sadaka, O. Levy, and E. Yahav, "Structural language models of code," in *Proceedings of 37th International Conference on Machine Learning*, 2020.

[59] A. See, P. J. Liu, and C. D. Manning, "Get to the point: Summarization with pointer-generator networks," in *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics*, 2017, pp. 1073–1083.

[60] Y. Shibata, T. Kida, S. Fukamachi, M. Takeda, A. Shinohara, T. Shinohara, and S. Arikawa, "Byte pair encoding: A text compression scheme that accelerates pattern matching," 1999.

[61] "Random forest by scikit-learn," http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html, 2021.

[62] G. Viviani, M. Famelis, X. Xia, C. Janik-Jones, and G. C. Murphy, "Locating latent design information in developer discussions: A study on pull requests," *IEEE Transactions on Software Engineering*, 2019.

[63] A. Alshammari, C. Morris, M. Hilton, and J. Bell, "Flakeflagger: Predicting flakiness without rerunning tests," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, 2021, pp. 1572–1584.

[64] F. Wilcoxon, "Individual comparisons by ranking methods," *Biometrics Bulletin*, vol. 1, no. 6, pp. 80–83, 1945.

[65] J. Zhang, H. Hong, Y. Zhang, Y. Wan, Y. Liu, and Y. Sui, "Disentangled code representation learning for multiple programming languages," in *Findings of the Association for Computational Linguistics: ACL-IJCNLP 2021*, 2021, pp. 4454–4466.

[66] N. R. Carvalho, J. J. Almeida, P. R. Henriques, and M. J. Varanda, "From source code identifiers to natural language terms," *Journal of Systems and Software*, vol. 100, pp. 117–128, 2015.

[67] Y. Jiang, H. Liu, Y. Zhang, N. Niu, Y. Zhao, and L. Zhang, "Which abbreviations should be expanded?" in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 578–589.

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

[69] G. Rodríguez-Pérez, G. Robles, and J. M. González-Barahona, "Reproducibility and credibility in empirical software engineering: A case study based on a systematic literature review of the use of the szz algorithm," *Information and Software Technology*, vol. 99, pp. 164–176, 2018.

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

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

[72] S. Panichella, A. Panichella, M. Beller, A. Zaidman, and H. C. Gall, "The impact of test case summaries on bug fixing performance: An empirical investigation," in *Proceedings of the 38th International Conference on Software Engineering*, 2016, pp. 547–558.

[73] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation with hybrid lexical and syntactical information," *Empirical Software Engineering*, vol. 25, pp. 2179–2217, 2019.

[74] Y. Zhou, R. Gu, T. Chen, Z. Huang, S. Panichella, and H. Gall, "Analyzing apis documentation and code to detect directive defects," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, 2017, pp. 27–37.

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

[76] S. Nguyen, H. Phan, T. Le, and T. N. Nguyen, "Suggesting natural method names to check name consistencies," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, p. 1372–1384.

[77] J. He, L. Xu, M. Yan, X. Xia, and Y. Lei, "Duplicate bug report detection using dual-channel convolutional neural networks," in *Proceedings of the 28th International Conference on Program Comprehension*, 2020.

[78] F. Liu, G. Li, B. Wei, X. Xia, M. Li, Z. Fu, and Z. Jin, "A self-attentional neural architecture for code completion with multi-task learning," in *Proceedings of the 28th International Conference on Program Comprehension*, 2020.

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

[80] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "A general path-based representation for predicting program properties," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2018, pp. 404–419.