Automated Comment Update: How Far are We?

Bo Lin*, Shangwen Wang*, Kui Liu†‡, Xiaoguang Mao*, Tegawendé F. Bissyandé§
*National University of Defense Technology, Changsha, China, {linbo19, wangshangwen13, xgmao}@nudt.edu.cn
†Nanjing University of Aeronautics and Astronautics, Nanjing, China, kui.liu@nuaa.edu.cn
‡State Key Laboratory of Mathematical Engineering and Advanced Computing, Wuxi, China
§University of Luxembourg, Luxembourg, tegawende.bissyande@uni.lu

***Abstract*—Code comments are key to program comprehension. When they are not consistent with the code, maintenance is hindered. Yet developers often forget to update comments along with their code evolution. With recent advances in neural machine translation, the research community is contemplating novel approaches for automatically generating up-to-date comments following code changes. CUP is such an example state-of-the-art approach whose promising performance remains however to be comprehensively assessed. Our study contributes to the literature by performing an in-depth analysis on the effectiveness of CUP. Our analysis revealed that the overall effectiveness of CUP is largely contributed by its success on updating comments via a single token change (96.6%). Several update failures occur when CUP ignores some code change information (10.4%) or when it is otherwise misled by additional information (12.8%). To put in perspective the achievements of CUP, we implement HEBCUP, a straightforward heuristic-based approach for code comment update. Building on our observations on CUP successful and failure cases, we design heuristics for focusing the update on the changed code and for performing token-level comment update. HEBCUP is shown to outperform CUP in terms of Accuracy by more than 60% while being over three orders of magnitude (i.e., 1 700 times) faster. Further empirical analysis confirms that the HEBCUP does not even overfit to the empirical analysis set. Overall, with this study, we call for more research in deep learning based comment update towards achieving state-of-the-art performance that would be unreachable by other less sophisticated techniques.**

***Index Terms*—Comment Update, Deep Learning, Empirical Assessment**

I. INTRODUCTION

Code comments constitute a key information channel in software development. They serve to publicize the intention behind a code fragment, record the design and implementation choices and provide information about to use the code [1], [2], [3]. Prior studies have shown that code comments play a significant role in improving program readability [4], [5] as well as facilitating the communication between developers [4], [6], [7], which are essential for ensuring program comprehension. Although the importance of code comments for code comprehension has been an important topic in the early days of software engineering [5], developers often fail to properly manage them. In particular, code comments are often forgotten when the code goes through changes [8], [9]. Therefore in practice, many comments are inconsistent or obsolete (hence bad) with respect to their associated code [8], [9], [10], [11]. For instance, Fluri *et al.* [12] empirically observed that newly added code is rarely commented and the percentage of comment changes that are triggered by source code changes can be rather low in some projects. In a recent study, Wen *et al.* [9] further pointed out that a substantial proportion of code-comment inconsistencies are introduced following refactoring activities.

TABLE I: A bad comment example.

Code Change:
public void shouldSucceedWhenAllowedSelfSigned() {
...
− IServiceConnector connector = factory.createConnector(request, service);
+ IApiConnector connector = factory.createConnector(request, api);
... }
Old Comment: gateway does not explicitly trust the service.
New Comment: gateway does not explicitly trust the API.

Table I overviews the real-world case example of a comment update in the APIMan project[1]. A commit changes the type of the *connector* being created from being to a Service into being to an API. Twice in the changed statement a token *service* is therefore replaced with *api*. Initially, however, the project developers (starting with the one who changed the code) forgot to update the comment associated with this method, leading to a case of inconsistent comment. Such bad comment can have a negative effect on program comprehension and may hinder maintenance [12], [13], [14]. Indeed, as supported by an analysis by Tan *et al.* [8], *bad comments* tend to introduce future project bugs. There is thus a dire need to fix bad comments, as it was later done for the method in Table I, or reduce the risk of keeping inconsistencies for a very long period.

Recently, Liu *et al.* [11] have proposed CUP as an automated just-in-time comment update technique to cope with the prevalence of bad comments. Their core idea is that inconsistencies can be avoided if comments are automatically updated with each code change. Leveraging neural machine translation advances and the wealth of big code, CUP was shown to achieve significant improvements over several baselines. In particular, CUP is promising in terms of capability to reduce developers' manual efforts for comments update. While this performance constitutes a literature milestone in comment update, little is known about where, how and why the proposed technique works or does not work. We fill this gap in the literature through a comprehensive assessment that analyses in-depth the strengths and weaknesses of the state-

*Shangwen Wang is the corresponding author.

[1]https://github.com/apiman/apiman

2643-7171/21/$31.00 ©2021 IEEE
DOI 10.1109/ICPC52881.2021.00013

of-the-art CUP technique. This assessment is performed on a carefully-cleaned dataset. In particular we ensured that the evaluation dataset does not include comment updates that are simply about improving language expression and not about the validity of the comment w.r.t. the code. We mainly find that:

- CUP generates comments by making trivial changes to existing comments. The vast majority (96.6%) of correct comments it generates are about modifying a single token.
- CUP generates correct comments when it can borrow the relevant content directly in the code change. In our experimental dataset such cases represent more than 90% of the correct comment updates.
- CUP frequently ignores some code change information while being often misled by other information. Thus, even when the content for updating a comment is available in the code change, CUP may fail.

Overall, we empirically showed that CUP generally fails when the update to the comment is not trivial, thus limiting its actual application scope. Besides this limitation in effectiveness, our experiments revealed that CUP is resource-intensive, which constitutes a severe obstacle to adoption since the practice of software engineering largely adheres to fast and straightforward approaches [15]. Towards further pointing out the necessity for sophisticated approaches to aim for higher levels of performance, we investigate the possibility of designing a simple but effective approach that would serve as a reasonable baseline on the task of automated comment update. We propose HEBCUP in this paper, which is a heuristic-based comment updater. Based on the findings in empirically analysing CUP generated comments, our design strategies are (1) focusing the update on the changed code and (2) performing token-level comment update. HEBCUP is a straightforward approach that implements a sequence of basic heuristics for updating comments based on code changes. Experimental evaluation results suggest that HEBCUP significantly outperforms CUP in terms of a variety of metrics (e.g., Accuracy, percentage of generated comments that are identical to human-written ones, etc). HEBCUP is furthermore over 1 700 times faster than CUP. Finally, our post-study experiments show that HEBCUP can generalize well on training and validation sets, achieving comparable Accuracy (23.8% and 25.5%, respectively).

To summarize, we make the following contributions:

- We perform comprehensive assessment and in-depth analysis of the state-of-the-art deep learning based comment updater. Our analysis findings provide insights into future work in this direction, highlighting the limitations that must be addressed towards making automated comment update research output that is valuable for practitioners.
- We also propose a basic approach to automated comment update as a baseline for the community. Since the performance of HEBCUP (baseline) exceeds that of CUP (state-of-the-art), we call for more research that will further explore the power of deep learning for the task of automatically updating code comments.

II. BACKGROUND

A. Comment UPdater

CUP is a recent-proposed deep learning-based automated comment updater [11]. Given the pre- and post-change versions of a code snippet (i.e., Java method) and its associated pre-change comment, CUP proposes to automatically generate a consistently-updated comment. CUP implements a generic neural sequence-to-sequence (seq2seq) model to learn the comment update patterns. Nonetheless, it presents several specialized designs for this task: (1) to deal with out-of-vocabulary (OOV) words, the authors propose a simple but effective way to tokenize code and comments while also being able to keep the format information of comments; (2) to capture relationships between code changes and comments better, the authors build a unified vocabulary for both code and comment tokens and integrate a novel co-attention mechanism to the model for effectively linking and fusing information in code changes and comments. Evaluation on thousands of popular Java projects from GitHub demonstrates the effectiveness of CUP: it can replicate comment updates performed by developers in a considerable number of cases (16.7%), and further analyses show that it can reduce developers' effort in comment updates.

B. Terminologies

We define several concepts that we refer to in this paper. We recall here that the state-of-the-art code embedding techniques [16], [17], [18] as well as CUP [11] are applied at the sub-token level: i.e., code tokens are broken into sub-token sequences based on camel case and under-score naming conventions. Prior works have empirically confirmed that doing so helps to better capture semantic information from source code [16], [19], [20].

- **[Edit distance]** (ED) denotes the minimum number of single-character edit operations required to transform a string into another one. Here, the considered operations include insertion, deletion, and replacement. For instance, the ED between "kitten" and "sitten" is 1 in that replacing "k" with "s" can fulfill the transformation.
- **[Number of changed sub-tokens]** (NCS) measures the number of sub-tokens that are modified before and after a code (comment) change. In the original study [11], the authors introduce an approach to convert a code change into an edit sequence via aligning its two sub-token sequences. For each aligned sub-token pair, an edit action (a_i) is assigned to indicate how to transfer the old code to the new one, which can be insert, delete, replace, or equal (denotes the two sub-tokens are identical). More details can be found in Section 3.1.2 of the paper [11]. Based on the obtained edit action sequence, the NCS of a code change (NCS_{cd})/comment change (NCS_{cm}) is defined as the sum of the numbers of insert, delete, and replace in the sequence. An example of code change is illustrated in Fig. 1 where the NCS_{cd} is 3 in that there are two inserts and one delete in the edit action sequence.

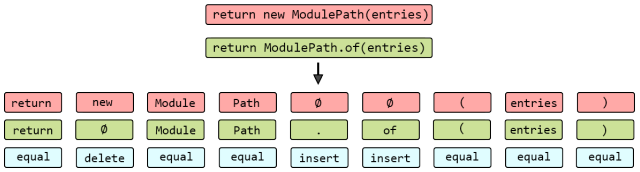


Fig. 1: The edit sequence of a code change.

- **[Single token update]**, a comment update whose changed content is only about one token.
- **[Single sub-token update]**, a comment update whose changed content is only about one sub-token. Note that this type of comment is a sub-type of the above one.
- **[Code-indicative update]**, a comment update whose changed sub-tokens or tokens can be found from the corresponding code change content.

III. Study Design

A. Data Cleaning

To evaluate the performance of CUP, the authors built a dataset from 1 496 repositories, which contains 85 657, 9 475, and 9 673 method-comment co-change instances respectively for training, validation, and test sets [11]. This is a large-scale, carefully-crafted, and open-sourced dataset, hence, we choose to reuse it in our study. Nevertheless, as observed by the authors of CUP, the dataset contains some instances where the comment updates only optimize the language expression (i.e., the old and new comments are of the same meaning). Such cases may bring bias to the evaluation since the comment update is not actually required. We thus propose to try our best to remove such cases from the dataset. We design the following rules to automatically identify three types of syntactic optimizations of the comments, which are *changing case of the word* (e.g., "capital" → "Capital"), *lexical translation* (e.g., pluralizing a verb or converting a noun into its verb form), as well as *fixing typos*:

- **[Rule-1]:** We compare the lowercase of the modified token before and after the comment update. If they are identical, the instance is identified as *changing case of the word*.
- **[Rule-2]:** We compare the root format of the modified token before and after the comment update with the help of the *lemmatization* function from NLTK tool[2]. If they are identical, the instance is identified as *lexical translation*.
- **[Rule-3]**: An instance is considered as *fixing typos* if it simultaneously satisfies the following conditions: (1) the edit distance (ED) between the pre- and post-updated tokens in the comment is less than or equal to 2; and (2) the post-updated token in the comment does not occur in the code change. The first condition is according to our observation that a typo may happen because both missing a letter and the order of two consecutive letters is reversed. For instance, in the linked commit[3], developers change the word "defautl" to "default", where the ED between the two words is 2. Hence, we set the threshold of ED to be 2 here. The second condition guarantees that the comment change is not based on the code change.

An instance is removed from the dataset upon identified as one of the above three types and thus being disregarded from our study. Finally, our cleaned dataset contains 80 591, 8 827, and 9 204 method-comment co-change samples for training, validation, and test sets with discarding 6 183 instances totally.

Note that there are also some semantic optimizations beyond syntactic optimization, where the new comments re-organize the words but express the same meaning as the old one. We do not target these cases since it is challenging to identify them automatically.

B. Research Questions

- **RQ1.** *What are the characteristics of the cleaned dataset?* We provide an in-depth analysis of the dataset, to enable the community to better qualify the performance of automated comment updaters. For instance, we investigate whether method-comment co-change instances in the test set are simpler than those from training and validation sets. Is the complexity of the comment update strongly related with that of its corresponding code change? Answering such questions can not only validate the rationale of the dataset but also help to better understand the inner correlation between code change and comment update for guiding the design of our own approach (see Section V-A). Thus, in the first RQ, we aim to dissect the characteristics of the dataset.
- **RQ2.** *How well does* CUP *perform on the cleaned dataset?* This RQ validates whether CUP holds its promising performances on the cleaned dataset. We assess whether the noisy data significantly influences the performance of CUP. Based on the results, we go further to check the generality of CUP (e.g., whether it tends to be more effective on simple instances than complex ones).
- **RQ3.** *Where and why does* CUP *work?*
- **RQ4.** *Where and why does* CUP *fail?*

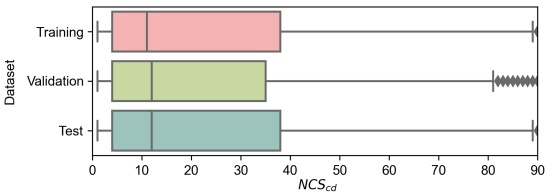
  These two RQs investigate the strengths and weaknesses of CUP. These questions are critical in that understanding the applicable scenario of an approach can better help us apply it in practice [21].

C. Evaluation Metrics

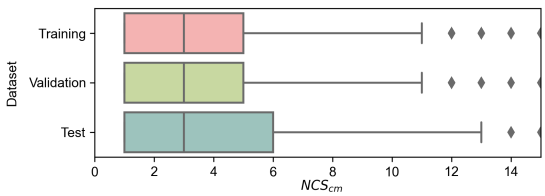
- **Accuracy:** the percentage of the test samples where *correct comments* are generated at Top-1. Here, *correct comments* refer to those that are identical to the ground-truth (i.e., written by developers).
- **Recall@5:** the percentage of the test samples where *correct comments* are generated at Top-5.
- **AED:** the average word-level Edit Distance required to change the predicted results from CUP into the ground-truth. This value indicates the distance between the generated comments and the ground-truth comments: the smaller, the better.

[2]https://www.nltk.org/.

[3]https://github.com/soot-oss/soot/commit/40d52c161164d391a0d3aad1d2fbc93e8a247dbd



(a) NCS_{cd}



(b) NCS_{cm}

Fig. 2: Distributions of NCS_{cd} and NCS_{cm} on three sets.

- **RED:** the average of the quotient of word-level ED required to change the predicted results from CUP into the ground-truth and word-level ED required to change the original comment into the ground-truth. This value indicates to what extent an approach can release developers' burden from manual update: the smaller, the better.

IV. Study Results

A. RQ1: [Dataset Characteristics]

In the original study [11], the authors split the dataset into training, validation, and test sets based on the ascending order of commit creation time within projects: the first 80% commits are put into the training set while the remaining 20% are split into the validation and test set randomly.

In this RQ, we first aim to investigate whether this process creates a balanced dataset by checking if instances of the test set tend to be simpler than instances from other sets (i.e., training and validation). We take NCS_{cd} and NCS_{cm} as metrics of complexity. The behind intuition is straightforward: a code change is more complex if it modifies more tokens and the same for comment change. We draw the distributions of these metrics in different datasets in Fig. 2.

From the perspective of code change, we observe similar distributions of NCS_{cd} among three sets: the lower quartiles are all 4 for the three groups, the medium value for training set is 11 while those of validation and test sets are both 12, and the upper quartiles are all around 35.

The same phenomenon can also be observed from the distributions of NCS_{cm} where the lower quartiles and medium values are 1 and 3 respectively for all three sets. The upper quartile of the test set (i.e., 6) is a bit higher than those of the other sets which are both 5.

We further perform a one-sided Mann-Whitney U-Test [22] on the distributions of three sets and results suggest that there is no significant difference between test set and the other two sets on either NCS_{cd} or NCS_{cm}, with p-values higher than 0.05 under all conditions. Hence, we conclude that our utilized dataset is balanced, that is, the instances from test set do not tend to be simpler than those from training and validation sets.

Finding-1 ☛ *Training, validation, and test sets in our dataset tend to possess similar distributions w.r.t NCS_{cd} and NCS_{cm}. Therefore, instances from the test set are not simpler than those from other two sets.*

Given that there are two complexity metrics in our study (i.e., NCS_{cd} and NCS_{cm}), a following question is whether these two variables possess correlation. In Fig. 3, we draw the scatter plot for cases from validation and test sets, where the horizontal axis denotes the value of NCS_{cd} and the vertical axis denotes the value of NCS_{cm}.

We observe the same phenomenon from these two sub-figures: the points distribute irregularly. The increase of NCS_{cd} does not necessarily lead to the increase of NCS_{cm} and vice versa. For instance, in the bottom right of Fig. 3a, there are two point whose x values are larger than 400 with y values only being 1. Such cases indicate that developers modify a large amount of sub-tokens in the code change while only modify 1 sub-token in the corresponding comment. We thus conclude that NCS_{cd} and NCS_{cm} possess no monotonic relationship. That is to say, one cannot predict NCS_{cm} based on NCS_{cd} and vice versa.

Note that we have analyzed the situation from training set and find a similar trend. We do not show the figure in this paper due to space constraints.

Finding-2 ☛ *The numbers of changed sub-tokens in code changes (NCS_{cd}) and in comment updates (NCS_{cm}) are not strongly related with each other.*

TABLE II: Effectiveness of CUP on Original and Cleaned Datasets.

Datasets	Accuracy	Recall@5	AED	RED
Original	16.7% (1 612/9 673)	26.1%	3.54	0.958
Cleaned	15.8% (1 456/9 204)	26.8%	3.62	0.960

B. RQ2: [Performance on Cleaned Dataset]

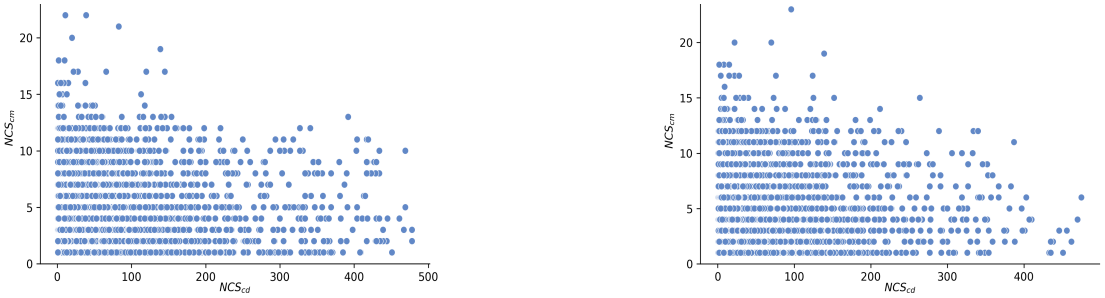
We re-train and re-evaluate CUP on our dataset on a server with 1660 Ti GPU. Results are listed in Table II.

We observe that the performance of CUP on the cleaned dataset is comparable with that from the original dataset. Although the accuracy drops slightly from 16.7% to 15.8%, the recall@5 increases from 26.1% to 26.8%, which means more ground-truth are generated in the top-5 candidates. Furthermore, the AED and RED values only experience negligible changes, e.g., , the AED changes from 3.54 to 3.62, meaning that the distance between the generated comment and the ground-truth is enlarged to a little extent.

This observation is in line with the one from the authors [11] which shows that CUP may also work poorly on the omitted language expression optimization cases.

Finding-3 ☛ *Switching to a cleaned dataset does not result in significant reduction of performance for* CUP.

To deeply understand the effectiveness of CUP, we further propose to check the generality of CUP, that is, whether CUP



(a) Cases from test set **(b)** Cases from validation set

Fig. 3: Scatter plot for cases from test and validation sets on their NCS_{cd} and NCS_{cm} values.

(a) NCS_{cd} **(b)** NCS_{cm}

Fig. 4: Comparison between the correct cases and the whole test set w.r.t NCS_{cd} and NCS_{cm}.

tends to generate correct comments for cases whose NCS_{cd} and NCS_{cm} are small. We compare the values of NCS_{cd} and NCS_{cm} from cases where correct comments are generated with figures from all cases in test set. The distributions are illustrated in Fig. 4. Note that the dataset *Correct* is hence a subset of the dataset *Test*.

This time, we observe obvious differences within the distributions. From the perspective of NCS_{cd}, the medium value of the *correct* dataset is 6 while that of the test dataset is 12. Similarly, from the perspective of NCS_{cm}, the upper quartile of *correct* dataset is 3, while that is only the medium value of the test set.

We again perform Mann-Whitney U-Test on the distributions from *correct* dataset and test dataset w.r.t NCS_{cd} and NCS_{cm}. Results reveal that the differences are significant with both p-values being less than 0.001. We thus reach the conclusion that CUP tend to generate correct comments for simple cases in the test set.

Finding-4 ☛ *The performance of* CUP *does not generalize: it is more effective for simple changes, e.g., where the number of sub-token changes for NCS_{cd} is small.*

C. RQ3: [Success of CUP]

So far we have known that CUP tend to make sense for simple cases measured by NCS_{cd} and NCS_{cm}. To deeply understand where and why CUP works, we manually analyze the 1 456 cases where it generates the correct comments in our evaluation. Two authors analyzed these cases independently and cross-checked each other's results until they reached consensus. Note that to provide more observations, we also analyze the results from the granularity of token in this and the following RQs.

Based on the manual analysis, we find that the success of CUP is mainly based on two factors which are: (1) the complexity of the comment change and (2) the relation between the comment change and code change. Specifically, whether the comment change is single token based or single sub-token based and whether the sub-tokens of comment change occur in the corresponding code change can significantly influence the results of CUP. We summarize our investigation in Table III. Please note that we have defined these types of comment updates in Section II-B.

TABLE III: The performance of CUP on different types of comment updates.

Comment update type	# U	# CU	Accuracy	# CIU	# C & CIU	Proportion
Single token	4426	1406	31.8%	3028	1347	95.8%
Single sub-token	2558	711	27.8%	1639	697	98.0%
All (test set)	9204	1456	15.8%	3639	1362	93.5%

U: update, **CU:** correct update, **CIU:** code-indicative update, **C & CIU:** correct and code-indicative update, **Accuracy:** number of CU divided by number of U, **Proportion:** number of C & CIU divided by number of CU.

Results reveal a number of findings. First, we find CUP is far more effective on single token/sub-token updates than on all cases. Its accuracy reaches 31.8% and 27.8% respectively, nearly twice as the average value on the whole test set. Furthermore, it can be calculated that nearly half (48.8% = 711/1 456) of correct comment updates are generated for *single sub-token update* and a huge amount (96.6% = 1 406/1 456) of correct comment updates are generated for *single token update*. It should be noted that a *single sub-token update* is also a *single token update* but not vice versa. That is the reason why the number of the former is less than that of the latter.

Finding-5 ☛ CUP *performs better on single token/sub-token updates than other cases. Single token/sub-token updates occupy a large amount of correct updates generated by* CUP *(96.6% and 48.8% respectively).*

In the right part of the table, we demonstrate some statistics about *code-indicative updates*. There are totally 3 639 code-indicative updates in the test set while the figures for single token/sub-token updates are 3 028 and 1 639 respectively. We note that code-indicative updates are more common in single token/sub-token updates. For instance, 68.4% (3 028/4 426) single token updates are code-indicative while the proportion

TABLE IV: Test sample 1.

Code Change:
 	
public synchronized int requestUpdate() {
this.needUpdate = true;
− return this.version;
+ return this.updateVersion;
}

Old Comment: Request an update of the current cluster metadata info, return the current version before the update.
New Comment & CUP: Request an update of the current cluster metadata info, return the current updateVersion before the update.

of all cases in test set is only 39.5% (3 639/9 204).

We do the statistics about the comment updates which are both code-indicative and correctly generated. We calculate the proportions of these updates in the correctly generated updates and list them in the last column. We find this time, no matter whether being single token/sub-token update or not, the proportions of code-indicative updates in the correctly generated cases are all extremely high (figures under all situations exceeding 90%). This suggests that for most of the successful cases, the relevant content for performing the comment update can be found in the corresponding code change.

Finding-6 ☛ *The vast majority of correct comment updates generated by* CUP *are code-indicative ones, either single token/sub-token updates or not.*

As shown in the table, among all the 3 639 code-indicative updates, CUP successfully captures code change information and generates correct comments for 1 362 cases. To better understand how CUP succeeds, we perform a case study on the code change listed in Table IV. With the change updating the return value of this method, developers update the document by changing the word token "version" to "updateVersion" to keep consistent. This is a single token update but not a single sub-token case since CUP is case-sensitive (the NCS_{cm} here is thus two: inserting "update" and replacing "version" with "Version"). CUP generates correct comment in this case.

We recall that CUP generally adopts a seq2seq paradigm, in which the new comment is generated sub-token by sub-token. When generating the following sub-token, CUP calculates probabilities for sub-tokens from three sources, which are (1) the whole vocabularies existing in the training set, (2) the old comment, and (3) the new method code. We draw in Fig. 5 the attention map of CUP when generating this new comment, which shows the relationship between the sub-token sources (vertical axis) and the generated sub-tokens (horizontal axis). The contribution of each sub-token source is the maximum weight from its contained sub-tokens. The color scale, shown on the right of the figure, varies from 0 (white) to 1 (black) and indicates the contribution of each sub-token source for generating an output sub-token. We note that under most conditions, CUP generates the sub-token by referring to the old comment. Nevertheless, as highlighted, when generating the two changed sub-tokens, information from new code plays a more important role. We manually checked the sources of the maximum weights in the new code for these two sub-tokens

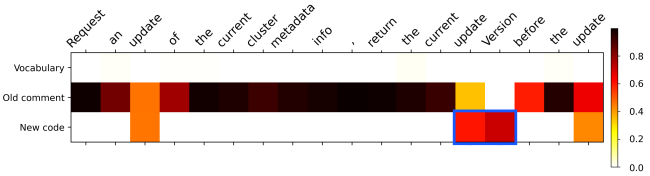


Fig. 5: Attention map for the case from Table IV generated by CUP.

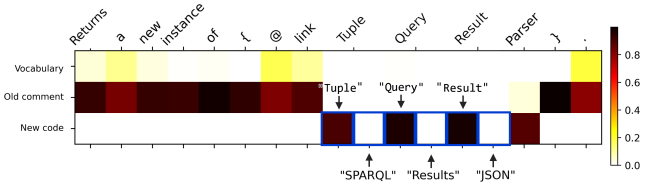


Fig. 6: Attention map for the case from Table V generated by CUP.

TABLE V: Test sample 2.

Code Change:
 	
− public RDFParser getParser() {
+ public TupleQueryResultParser getParser() {
return new BigdataSPARQLResultsJSONParser();
}

Old Comment: Returns a new instance of {@link TurtleParser}.
New Comment: Returns a new instance of {@link SPARQLResultsJSONParser}.
CUP: Returns a new instance of {@link TupleQueryResultParser}.

and confirmed that they are indeed the changed sub-tokens (i.e., "update" and "Version"). This indicates that CUP can indeed capture the code change information and utilize it for guiding the generation of new comment.

Finding-7 ☛ CUP *can capture code change information well for around 37% of code-indicative updates (1 362/3 639).*

D. RQ4: [Failures of CUP *]*

To investigate where and why CUP fails, we conduct another manual analysis. This time there is totally 7 748 cases. Thus, it is challenging, if not impossible, to analyze all of these cases manually. We thus randomly sample one thousand for manual analysis. Again, two authors analyzed them and checked the results before reporting the following findings.

We find that the major reason for the failure of CUP is the exclusion of comment update content in the code change, happening in 70.6% cases (706/1 000) of our manual analysis. An example is given in Table V. The newly added sub-tokens in the comment (i.e., *SPARQL*, *Results*, and *JSON*) do not appear in the code change but in the unchanged part of the code. However, CUP generates the comment based on the changed part of the code. We draw the attention map in Fig. 6. Note that to make clear comparison between the newly-added code sub-tokens and the oracles (i.e., sub-tokens that should be appeared in the comment), we split the grid at the corresponding locations and illustrate the weights of newly-added sub-tokens and oracles at the left and right sides, respectively. From the results, the weights of newly-added code sub-tokens are much higher than those from the oracle

TABLE VI: Test sample 3.

Code Change:

− public PartBarline getRightBarline ()
+ public PartBarline getRightPartBarline ()
{
return rightBarline;
}

Old Comment: Report the ending Barline.
New Comment: Report the ending PartBarline.
CUP: Report the ending Barline.

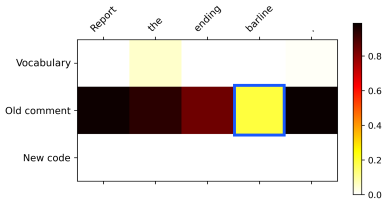


Fig. 7: Attention map for the case from Table VI generated by CUP.

TABLE VII: Test sample 4.

Code Change:

− public boolean isReprUrlDirty(java.lang.CharSequence value) {
+ public boolean isHeadersDirty() {
− return isDirty(17);
+ return isDirty(18);
}

Old Comment: Checks the dirty status of the 'ReprUrl' field.
New Comment: Checks the dirty status of the 'Headers' field.
CUP: Checks the dirty status of the '18' field.

sub-tokens which are close to zero, leading to the wrong result outputted by CUP. For instance, when generating the next sub-token after *link*, the weight of the newly-added code *Tuple* is more than 0.8, much higher than that from the oracle sub-token *SPARQL* whose attention weight is almost 0. The same phenomenon can be observed for the following two sub-tokens. Such a case indicates that sometimes CUP may over-emphasize the information from code change while ignoring the other part of code.

There are also some code-indicative updates that cannot be generated by CUP. We find mainly two reasons for such situations. The first is that CUP does not capture the code change information and still generates the old comment, happening in 10.4% of cases (104/1 000). An example is illustrated in Table VI where developers modify the method name, and thus the comment should be updated consistently. Unfortunately, CUP does not update it successfully but generates the old comment. The corresponding attention map in Fig. 7 demonstrates the generation process of CUP. We note that when it is expected to generate the sub-token *Part*, CUP focuses on information from the old comment which is actually *barline*, leading to its failure. Such a case indicates that sometimes the information from code change can be ignored by CUP.

The second reason is that in 12.8% of cases (128/1 000), it is also possible for CUP to be misled by the additional information in the code change. We give an example in Table VII. In this case, there are two changed sub-tokens in the new code which are *Headers* and *18*. When updating the comment, developers only concentrate on the filed name reflected by the method name so that the parameter *18* is

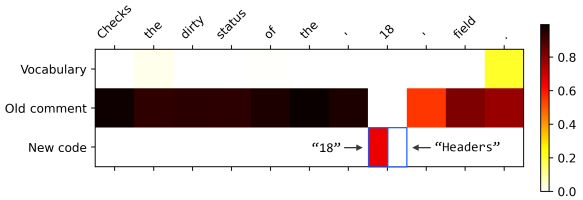


Fig. 8: Attention map for the case from Table VII generated by CUP.

useless information. However, CUP does not generate the correct update. Instead, it includes *18* in its output. The attention map in Fig. 8 demonstrates the generation process. The rewarding thing is that CUP does know where to update, i.e., it modifies the correct sub-token. However, it generates a wrong sub-token. As we can see, the weight of *18* is much higher than that of *Headers*, which is almost 0.

> **Finding-8** ☛ CUP *often fails mainly because (1) it cannot borrow the relevant content directly in the code change, (2) code change information is unfortunately ignored (10.4%), and (3) it is misled by the additional information in the code change (12.8%).*

V. Heuristic Based Alternative Approach

CUP is a sophisticated DL-based approach that requires training on a large corpus of high quality method-comment co-change instances. It is quite time-consuming and resource-consuming to learn such its final generation model. Furthermore, its inner working is hard to interpret. Specifically, it sometimes captures the code change information (cf. Fig. 5), and sometimes exaggerates the role of code change information (cf. Fig. 6). In other cases, this information is ignored (cf. Fig. 7). In this section we investigate whether a simple and straightforward approach can provide reasonable performance for code comment update. We refer this approach as HebCup, an heuristic-based comment updater.

The design strategy of HebCup is based on our empirical findings that (1) nearly all correctly generated updates are code-indicative ones (**Finding-6**) and (2) more correct comment updates are generated at the token level than the sub-token level (cf. Table III). We are thus motivated to concentrate on the changed code as well as perform the comment update at the token level.

A. Proposed Approach

The pipeline of HebCup is demonstrated in Fig. 9. Given a pair of statements before and after a code change, we first identify the changed tokens. Then, for each modified token, we align its sub-tokens to capture the added/deleted/replaced sub-tokens, after which we construct token-level replacement pairs (i.e., the mapping relations between old tokens and their potential new tokens for update) based on the modified sub-tokens. The intuition is that the changed code token may not appear in the comment. For instance, if a method name is changed from `getX` to `getY`, its comment may be updated from "return X" to "return Y", where the update object cannot

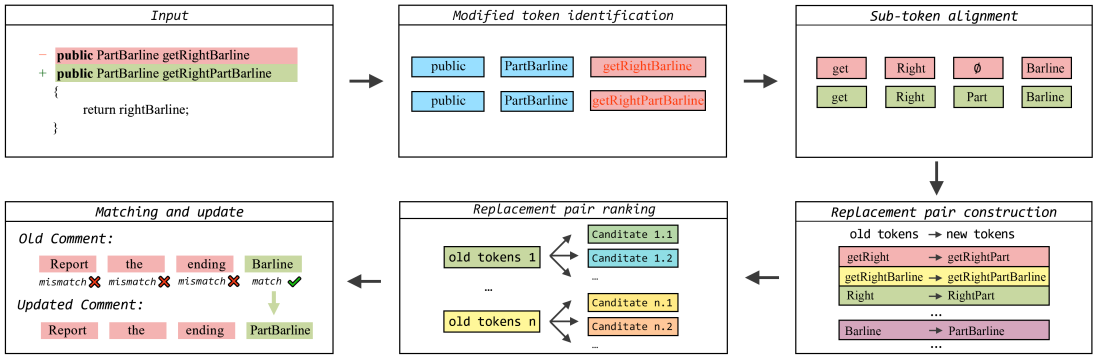


Fig. 9: Overview of the HEBCUP for comment update.

be found from the code change through token-level analysis. Hence, directly using the pre- and post-updated code token for comment update is inappropriate. We are thus motivated to extract more information via analyzing sub-token level differences. In case an old token corresponds to several new tokens, we further assign each candidate a priority. In the last step, we seek if there is any old token that appears in the old comment. Upon identification, we update it using its corresponding new token.

Modified token identification. Given a code pair containing the old and new statements as input, we tokenize the statements and locate the modified tokens. This is done through the lexer utilized in CUP.

Sub-token alignment. The inputs of this step are two different tokens, one from the old code while another from the new code (*getRightBarline* and *getRightPartBarline* in our example). The step aims to find out the changed sub-tokens and align these two tokens for the next step.

Algorithm 1 details the process. Given two tokens, we first compare their first sub-tokens to check if the two sub-tokens are identical (cf. line 3). If so, the mapping relation between the two sub-tokens is recorded and we iteratively compare the rest of the two tokens (cf. lines 4 and 5). If not, we start from the tail to check if the two last sub-tokens are identical and perform a similar operation (cf. lines 7 to 9). This iteration process is terminated when (1) there is no element in the input old/new sub-token list which indicates an addition/deletion operation or (2) neither the head sub-tokens nor the tail sub-tokens can be matched which indicates a replacement operation. Under the former condition, we use ϕ to map with the left of the token which still has content (cf. lines 11 to 16); while under the latter condition, the left parts of the two tokens are mapped directly (cf. lines 17 to 20). For instance, for the case in Fig. 9, the two tokens are aligned in the order of $get \rightarrow get$, $Right \rightarrow Right$, $Barline \rightarrow Barline$, $\phi \rightarrow Part$. Note that we also restore the order of the mapping relations based on the original tokens. Hence, the output of this step is two ordered and aligned sub-token sequences.

Replacement pair construction. So far we have identified the modified sub-token in the code (i.e., the different part in the two input sequences). In this step, we construct token-level

Algorithm 1: Sub-token alignment algorithm.

Input: The old token T_O with sub-token sequence s_{old} = { SO_1, SO_2, ..., SO_m }.
Input: The new token T_N with sub-token sequence s_{new} = { SN_1, SN_2, ..., SN_n }.
Output: The map that records how the two sequences are aligned.

```
1  Function SA(map, s_old, s_new):
2      /* Align from the head.                          */
3      if s_old.first == s_new.first then
4          map.add(s_old.first → s_new.first) ;
5          SA(map, s_old.removefirst, s_new.removefirst) ;
6      /* Align from the tail.                          */
7      if s_old.last == s_new.last then
8          map.add(s_old.last → s_new.last) ;
9          SA(map, s_old.removelast, s_new.removelast) ;
10     /* If one of the sequences is null, pad it
          with φ to align with the other one.          */
11     if s_old.length == 0 then
12         map.add(φ → s_new) ;
13         return map ;
14     else if s_new.length == 0 then
15         map.add(s_old → φ) ;
16         return map ;
17     else
18         /* If no sub-token can be matched, the
              left sub-tokens are mapped as a
              whole.                                   */
19         map.add(s_old → s_new) ;
20         return map ;
```

replacement pairs for comment update, which is to enumerate all possible old tokens with their potential updates.

Formally, given two aligned sub-token sequences, st_o and st_n, with length being l. The replacement pairs rps are:

$$rps = \{< st_o[i:j], st_n[i:j] > | \ 1 \leq i \leq j \leq l\}$$

where $st_o[i:j]$ denotes the token compromised of the i_{th} through the j_{th} sub-tokens of st_o and the similar for $st_n[i:j]$. Note that we directly ignore ϕ when connecting sub-tokens to shape tokens. The output set of this step can thus be considered as a HashMap where the *keys* are old tokens and *values* are potential new tokens for replacement. Pairs whose *keys* and *values* are identical are not updates and are thus removed.

Replacement pair ranking. A code change may modify several statements in the method. As a result, in practice, an old token may possess several candidate new tokens for replacement. For instance, in a changed method from the

TABLE VIII: Effectiveness of HEBCUP vs. CUP.

Technique	Accuracy	Recall@5	AED	RED
CUP	15.8% (1 456/9 204)	26.8%	3.62	0.960
HEBCUP	25.6% (2 360/9 204)	27.6%	3.52	0.896

TABLE IX: Efficiency of HEBCUP vs. CUP.

Technique	Device	Training Time	Testing Time
CUP	GTX 1660 Ti	8 hours	22 mins
HEBCUP	CPU	N/A	17 secs

GTX 1660 Ti refers to Nvidia GeForce GTX 1660 Ti. CPU is Intel Core i7 3.0GHz.

linked commit [4], the token *read* possesses two candidates which are *hasReadPermission* and *readPermission*. We thus, in this step, sort these candidates according to the number of sub-tokens they own. The behind intuition is that a token with more sub-tokens can possess more semantic information and hence is more likely to appear in the comment for program comprehension. After this step, the *values* (new tokens) for the same *keys* (old tokens) in the replacement pair set are ranked.

Matching and update. The comment is updated in the last step. Since NCS_{cd} and NCS_{cm} are not strongly related (**Finding-2**), one cannot infer NCS_{cm} from NCS_{cd}. Hence, we traverse tokens in the old comment and if any of them matches the *keys* in the replacement pair set, we use the corresponding *values* to update it (which means we do not set a pre-defined threshold of NCS_{cm}). The update process ends after the traversing, therefore, HEBCUP can deal with both single token updates and multiple token updates. If no matched token found, HEBCUP outputs the original comment.

B. Evaluation

We compare our HEBCUP against the state-of-the-art CUP w.r.t. effectiveness and efficiency on our cleaned dataset. Results on effectiveness are shown in Table VIII. To calculate recall@5 of HEBCUP, we allow it to try five candidates for each matched old token. Results reveal that our approach outperforms CUP on all metrics. Specifically, its accuracy value reaches 25.6%, exceeding that of CUP by around 62%. The RED value of HEBCUP decreases to lower than 0.9, which indicates that it can better help reduce the manual effort.

Results on efficiency are shown in Table IX. We can see that it takes 8 hours to train CUP and 22 minutes to test on the cleaned dataset. Since HEBCUP does not need training, its training time is marked as "N/A". The time cost of its testing process is only 17 seconds. This means that HEBCUP is considerably (more than 1 700 times) faster than CUP.

Compared against CUP*,* HEBCUP *provides better performance in automated comment updates, in terms of all metrics.* HEBCUP *is furthermore about 1 700 times more efficient than* CUP*, on our cleaned dataset.*

VI. DISCUSSION

A. Generalisation of HEBCUP

There is a concern that our designed HEBCUP is overfitting to the test set where we perform this empirical study. To assess its generality, we perform a post-study experiment to evaluate HEBCUP on other parts of our dataset. Specifically, we evaluate HEBCUP on the validation set. For comparison, we also re-train CUP with exchanging the purposes of the original validation and test sets. Results are shown in Table X. We obtain two insights, both of which demonstrate the generality of HEBCUP. First, HEBCUP still outperforms CUP with respect to all metrics. Second, the performance of HEBCUP declines little compared against that on the test set, even achieving a better AED. Furthermore, we also evaluate HEBCUP on the training set. The accuracy value can also reach 23.8% (19 214/80 591), which further indicates that the performance of HEBCUP is not restricted to the test set. Such results are consistent with the fact that the proportions of code-indicative updates are similar among the three sets, which are 35.3% (28 426/80 591), 39.8% (3 517/8 827), and 39.5% (3 639/9 204) in training, validation, and test sets respectively.

[4]https://github.com/Azure/azure-sdk-for-java/commit/7411f8e0d721f43c4be9e7e797edc08af5e7093d.

TABLE X: Effectiveness of HEBCUP vs. CUP/ validation set.

Technique	Accuracy	Recall@5	AED	RED
CUP	19.3% (1 702 /8 827)	26.0%	3.58	0.963
HEBCUP	25.5% (2 251/8 827)	27.2%	3.31	0.960

B. Implications

Simple approaches first. Our study suggests that fast and straightforward approaches may achieve comparable or even better performances on the specific software engineering task, comment update, compared with time-consuming deep learning techniques. This finding supports a general call by Fu and Menzies [15] to "try-with-simpler" approaches that capture the characteristics of the data while incurring limited computing cost. Nevertheless, we greatly thank Liu *et al.*'s efforts to design CUP, without which designing a simple comment updater in the first place might be difficult.

Long way to go for comment update. Although HEBCUP achieves significant improvements on CUP, it should be noted that it still can only work on code-indicative updates. It is difficult to manually define templates for comments whose updated contents do not appear within the code change content. Similarly, despite being general to all cases theoretically, the vast majority of success cases from CUP, are also code-indicative ones. Such results indicate that in the future, mining the relationship between the code comment and the unchanged part during code change deserves more in-depth analysis. There is still a long way to go for automatically updating code comments.

C. Threats to Validity

A threat to external validity is related to the dataset we used. It is possible that CUP and HEBCUP demonstrate different performance on code-comment co-change cases that are excluded in this dataset. Nevertheless, this is mitigated in that our dataset contains instances from a large scale (i.e., 1 496) of top-starred GitHub projects.

A threat to internal validity is that we only randomly select 1 000 failure cases for dissecting the weaknesses of CUP. Consequently, some findings may be ignored due to

this reduction. Nonetheless, it is challenging to analyze all the cases manually and is thus left as our future work.

VII. Related Work

A. Studies on Code Comment

1) Comment Classification: While code comments help enhance the readability of the code, it is believed that not all the comments have the same goal and target audience [2]. Consequently, researchers propose to classify the code comments for better program comprehension. Haouari *et al.* [23] firstly proposed an initial taxonomy via manually defining four features,while Steidl *et al.* [3] defined seven different comment categories and then leveraged machine learning models for automatic comment classification. More recently, Pascarella *et al.* [2] performed a large-scale empirical study where a hierarchical taxonomy of Java code comments is built.

2) Comment Generation: One way that may help developers update comments is to generate comments for updated methods from scratch directly. Sridhara *et al.* [24] proposed an automatic approach that generates natural language summarization of the method's overall actions via manually defined templates. Hu *et al.* [25] designed an encoder-decoder model to generate comments by integrating source code with structure information. LeClair *et al.* [26] proposed another data-driven approach that also combines words from code with code structure. Note that our study subject in this paper, CUP, focuses on updating on-hand code comments rather than generating from scratch. It thus can be considered as dealing with another problem compared with these related works.

3) Inconsistent Comment Detection: Researchers have investigated the detection of inconsistent comments. Tan *et al.* [27] proposed an approach to infer program properties from source code and generate random tests to check the inferred properties. Ratol and Robillard [28] designed a rule-based approach to detect *fragile comments*, which denotes comments that become inconsistent during identifier renaming. Zhou *et al.* [29] utilized constraint solver to detect defects from directives of the API documents.

4) Comment Usage: Researchers also pay attention to how to utilize high-quality comments. Eberhart *et al.* [30] designed an automated approach to extract summary descriptions of subroutines from unstructured code comments. Tan *et al.* [31] extracted interrupt related annotations from code and comments for detecting operating system concurrency bugs. Farias *et al.* [32] reduced false positives generated by self-admitted technical debt (SATD) detection techniques via code comment analysis.

B. Deep Learning for Program Comprehension

Recently, researchers propose to solve program comprehension tasks by taking the advantages of big data and deep learning techniques. Iyer *et al.* [33] presented the first data-driven approach for generating high level summaries of source code. Recently, LeClair *et al.* [34] obtained the state-of-the-art performance on this task by separately encoding structure information and source code sequence. Deshmukh *et al.* [35] first proposed to detect duplicate bug reports with Convolutional Neural Networks (CNN) and Long Short Term Memory (LSTM). He *et al.* [36] further used a dual-channel CNN model to represent a bug report pair together. Li *et al.* [37] proposed a model mixed with LSTM and pointer network for code completion. Liu *et al.* [38] solved this task via capturing both structure information and long-term dependency of the input programs.

C. Machine Learning vs. Heuristics

Some retrospective studies discuss whether the advanced deep learning techniques really outperform traditional approaches. To generate commit messages, Jiang *et al.* [39] adopted a neural machine translation (NMT) technique to learn from code change *diffs*, which is indeed technical sound since it does not require manually defined templates. Nonetheless, Liu *et al.* [21] performed an in-depth analysis on the performance of this technique and found that a simple approach based on the nearest neighbor algorithm can better generate commit messages from *diffs*. Jiang *et al.* [40] analyzed where and why the state-of-the-art AST path-based code representation technique, code2vec [20], works on method name generation. Based on their findings, they designed a heuristic which is rather simple but can outperform code2vec significantly. Similarly, Pecorelli *et al.* [41] conducted a large-scale study to compare the performance of heuristic-based and machine-learning-based techniques for metric-based code smell detection and found that heuristic techniques perform better. Our results also demonstrate that a simple approach may outperform complex neural networks in specific tasks (e.g., comment update).

VIII. Conclusion

With recent advances in deep representation learning techniques, several tasks in software engineering are beeing revisited by the community towards implementing automation. Automated comment updating is one such task. In this paper, we perform an empirical study on CUP, a recent state-of-the-art DL-based comment updater. We investigate where and why it works or fails. Based on our findings, which highlight that the gaps that research must fill, we propose a simple heuristic-based method as a baseline for future work. HebCup was found to be both more effective and more efficient than CUP. Like CUP, however, HebCup is not sufficiently effective beyond simple comments. All data in the study are publicly available at: **https://github.com/Ringbo/HebCup**.

Acknowledgements

This work was supported by the Open Project Program of the State Key Laboratory of Mathematical Engineering and Advanced Computing, the National Natural Science Foundation of China (Grant No.61872445, No.61672529 and No.61802180), the Natural Science Foundation of Jiangsu Province (Grant No.BK20180421), the National Cryptography Development Fund (Grant No.MMJJ20180105) and the Fundamental Research Funds for the Central Universities (Grant No.NE2018106).

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

[5] T. Tenny, "Program readability: Procedures versus comments," *IEEE Transactions on Software Engineering*, vol. 14, pp. 1271–1279, 1988.

[6] 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.

[7] M.-A. Storey, J. Ryall, R. lan 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.

[8] 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.

[9] 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.

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

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] W. Fu and T. Menzies, "Easy over hard: a case study on deep learning," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017.

[16] 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.

[17] T. Hoang, H. J. Kang, J. Lawall, and D. Lo, "CC2Vec: distributed representations of code changes," in *Proceedings of the 42nd International Conference on Software Engineering*. ACM, 2020, pp. 518–529.

[18] 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.

[19] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, "Suggesting accurate method and class names," in *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering*. ACM, 2015, pp. 38–49.

[20] 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.

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

[22] 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.

[23] D. Haouari, H. Sahraoui, and P. Langlais, "How good is your comment? a study of comments in java programs," in *Proceedings of the 2011 International Symposium on Empirical Software Engineering and Measurement*, 2011, p. 137–146.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] Z. Eberhart, A. LeClair, and C. McMillan, "Automatically extracting subroutine summary descriptions from unstructured comments," in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2020, pp. 35–46.

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

[32] M. A. Farias, M. G. Mendonça, M. Kalinowski, and R. Spínola, "Identifying self-admitted technical debt through code comment analysis with a contextualized vocabulary," *Information and Software Technology*, vol. 121, 2020.

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

[34] A. LeClair, S. Haque, L. Wu, and C. McMillan, "Improved code summarization via a graph neural network," in *Proceedings of the 28th International Conference on Program Comprehension*, 2020.

[35] J. Deshmukh, K. M. Annervaz, S. Podder, S. Sengupta, and N. Dubash, "Towards accurate duplicate bug retrieval using deep learning techniques," in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2017, pp. 115–124.

[36] 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.

[37] J. Li, Y. Wang, M. R. Lyu, and I. King, "Code completion with neural attention and pointer networks," in *Proceedings of the 27th International Joint Conference on Artificial Intelligence (IJCAI)*, 2018.

[38] 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.

[39] S. Jiang, A. Armaly, and C. McMillan, "Automatically generating commit messages from diffs using neural machine translation," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, 2017, pp. 135–146.

[40] L. Jiang, H. Liu, and H. Jiang, "Machine learning based recommendation of method names: How far are we," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019, pp. 602–614.

[41] F. Pecorelli, F. Palomba, D. D. Nucci, and A. D. Lucia, "Comparing heuristic and machine learning approaches for metric-based code smell detection," in *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, 2019, pp. 93–104.