Towards Usable Neural Comment Generation via Code-Comment Linkage Interpretation: Method and Empirical Study

Shuyao Jiang, Jiacheng Shen, Shengnan Wu, Yu Cai, Yue Yu, and Yangfan Zhou

Abstract—Code comment is important to facilitate code comprehension for developers. Recent studies suggest to generate comments automatically with deep learning, in particular, based on neural machine translation models. However, such a promising Neural Comment Generation (NCG) technique suffers from unsatisfactory performance, as well as poor usability, i.e., developers cannot easily understand and modify the auto-generated comments. This paper suggests that a proper interpretation of how the comments are generated can significantly improve the usability of NCG approaches. We propose a novel model-independent framework, namely CCLink, to interpret the auto-generated comments. CCLink generates a set of code mutants and obtains their corresponding comments. Based on these data, several contribution mining algorithms are designed to infer the key elements in code that contributes to the generation of the key phrases in the comments. The links between code and its auto-generated comment can thus be constructed. This in turn allows CCLink to visualize the links as the comment interpretations to developers. It greatly facilitates manual verification and correction of the comments. We examine the performance of CCLink with different contribution mining algorithms, NCG approaches, and real-world datasets. We also conduct an empirical study on 32 experienced Java programmers to evaluate the effectiveness of CCLink. The results show that CCLink is promising in making NCG more usable with a proper interpretation of the auto-generated comments.

Index Terms—Code comment, neural comment generation, software usability

1 Introduction

Code comprehension is critical to modern software development, especially for open source projects that new developers frequently join. It is a rigid need for those new developers to understand the project code written by others, while it is also a quite labor-intensive task (e.g., 80% of the development time is spent on understanding code as reported in [1]).

It has long been accepted that code comments, usually brief descriptions of code snippets, can greatly facilitate code comprehension. However, original developers may not write good comments in practice due to negligence or time constraints [2], [3]. As a result, new developers have to put a lot of repetitive efforts to understand the existing code. A promising solution is automatic code comment generation [4], [5], which auto-summarizes code into short, natural language text with an aim to reduce human efforts in understanding the source code.

Current state-of-the-art approaches apply Neural Machine Translation (NMT) to perform this task [6], [7], [8], [9]. Specifically, they adopt sequence-to-sequence neural networks (e.g., [10]) to convert source code into natural language (i.e., code comments). Such *Neural Comment Generation* (NCG) approaches can learn from human experiences, i.e., high-quality mannually-written code comments generally available in mature open source projects (e.g., Linux, Android, OpenStack). With the recent advancement of deep learning, NCG is considered a very promising research direction [9].

Unfortunately, state-of-the-art NCG approaches are still far from being usable in practice [11]. Despite tremendous research efforts to improve NMT models, the quality of the generated comments is not yet satisfactory. For example, `DeepCom` [9] and `ast-attengru` [8], two state-of-the-art NCG approaches, generally cannot achieve a BLEU value over 40% (BLEU is a widely-used metric to evaluation text quality [12]). Such poor performance makes developers unwilling to use NCG approaches in practice.

Except for the unsatisfactory performance, the lack of good usability is also a critical reason why NCG is not applicable. It is difficult for developers to understand and modify the auto-generated comments. This work, unlike existing studies that emphasize on improving the performance of NCG, proposes to address its usability issue. We consider that the *interpretability* of the auto-generated comments is critical to the usability of NCG approaches. Actually, it has

- *Shuyao Jiang, Jiacheng Shen, Shengnan Wu, Yu Cai, and Yangfan Zhou are with the School of Computer Science, Fudan University, Shanghai 200437, China, and also with the Shanghai Key Laboratory of Intelligent Information Processing, Shanghai 201203, China. E-mail: {syjiang19, jcshen16, snwu19, caiy19, zyf}@fudan.edu.cn.*
- *Yue Yu is with the National Laboratory for Parallel & Distributed Processing, National University of Defense Technology, Changsha 410073, China. E-mail: yuyue@nudt.edu.cn.*

Manuscript received 5 January 2022; revised 8 October 2022; accepted 12 October 2022. Date of publication 17 October 2022; date of current version 18 April 2023.
This work was supported in part by the National Key R&D Program of China under Grant 2020AAA0103504, and in part by the Natural Science Foundation of Shanghai under Grant 22ZR1407900.
This work involved human subjects or animals in its research. The author(s) confirm(s) that all human/animal subject research procedures and protocols are exempt from review board approval.
(Corresponding author: Yangfan Zhou.)
Recommended for acceptance by L. Tan.
Digital Object Identifier no. 10.1109/TSE.2022.3214859

long been suggested that providing an interpretation of software behavior will significantly improve the software usability [13], [14], [15], [16], especially for software that adopts machine learning [17], [18] in a black-box manner. In the context where new developers would like to contribute comments to open source projects, proper interpretations of the auto-generated comments can greatly facilitate developers to comprehend and further improve the comments.

To this end, we propose a novel framework CCLink (Code-Comment Links) to interpret the comments generated by NCG approaches. CCLink aims to find the code segments which contribute to the generation of key information (i.e., key phrases called *focuses*) in the auto-generated comments. CCLink is inspired by the success of natural language translation (e.g., Google Translate [19]), where its result is instantly explainable: A user can check the resulting phrases against their counterparts in the source sequence. In this way, the quality of the translation can be easily perceived, and the user can accordingly modify the incorrect phrases. In addition, those modifications can be recorded and further used as a new type of valuable feedback data to refine the original intelligent models.

However, the link between code and its comment is less obvious. It is not straight-forward for a developer to know why a sentence is generated with the code. Automatically inferring such links is actually very challenging. It is notoriously difficult to interpret deep learning models [20]. Inferring such links by examining the underlying logic of an NCG approach is infeasible currently. Fortunately, we note that code is also a way of human communication [11], where its internal documentation (e.g., variable names) to some extent reflect its intention [21], [22]. This allows CCLink to mine the links between code and its auto-generated comment, by considering the deep learning model of an NCG as a black box.

In particular, CCLink first determines the *focuses* (i.e., key phrases) in the auto-generated comment, then infers which parts of code result in each *focus* with a *contribution mining* method. To this end, CCLink generates a series of code mutants of the code and obtains their auto-generated comments. Based on the resulting comments, CCLink divides the mutants into two categories: one can produce the comments with a specific *focus* and the other cannot. Based on these data, we tailor data mining algorithms to determine the code segments that contribute to the generation of the *focus*. The links between code and its auto-generated comment can thus be constructed. This in turn allows CCLink to visualize the links as the comment interpretations to developers.

To the best of our knowledge, CCLink is the first solution dedicated to improving the usability of NCG from the perspective of interpretability. We design comprehensive experiments to evaluate CCLink. First, we examine the performance of CCLink with different contribution mining algorithms, NCG approaches and real-world datasets. Then, we conduct an empirical study, where we recruit 32 experienced Java programmers to perform a comment-correction task with/without the interpretation functionality of CCLink. The results verify the effectiveness of CCLink. The source code of CCLink (including that for experiments) is open-source available at https://github.com/CCLink-demo.

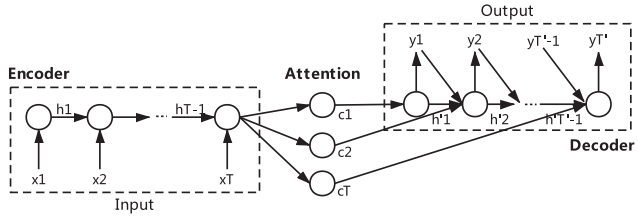


Fig. 1. Attentional RNN Encoder-Decoder.

The main contributions of this paper are as follows:

- We are the first to propose the interpretability of the auto-generated comments is key to the usability of NCG approaches, and provide a systematic study on comment interpretation.
- We design CCLink, a novel NMT model-independent framework to interpret auto-generated comments, which includes several specifically-tailored interpretation methods for this task.
- We show that CCLink is promising in making NCG more usable with a proper interpretation of the auto-generated comments via comprehensive experiments and user studies.

The rest of paper is organized as follows. Section 2 provides the preliminaries and our research motivation. We elaborate our CCLink design in Section 3. Sections 4 and 5 report our experimental results and empirical study. Section 6 further discusses our experimental and empirical studies, where we provide and analyze good/bad examples of links, the threats to validity, and the implications. We survey related work in Section 7 and finally conclude this work in Section 8.

2 Background and Motivation

Neural machine translation (NMT) is a machine learning technique that adopts deep neural networks to convert a source language sequence to a target language sequence. It has shown its good performance in language translation, without requiring feature extraction efforts typically required in traditional approaches [23], [24]. Fig. 1 shows a typical, widely-used NMT model as an example [10]. It applies *Attentional RNN Encoder-Decoder*, which includes two RNNs (recurrent neural networks). One RNN, namely, the encoder, transforms the source sequence to a vector representation (i.e., an embedding). The other, namely, the decoder, then transforms the embedding into the target sequence. An attention mechanism is usually introduced to determine the weights of the words in the source sequence during translation [10]. RNN cells can also be replaced with long short-term memory (LSTM) [25] or the gated recurrent unit (GRU) [26], so as to handle long-sequence scenarios.

NMT has recently been proposed to conduct automatic code comment generation [6], [27], which we call *Neural Comment Generation* (NCG) in our following discussion. The underlying notion of NCG is that software code is a form of human communication, which has similar statistical properties as natural language [11]. By treating code as a source language sequence, an NMT model can convert it into a target language sequence (i.e., code comments). Existing studies have shown that NCG is an effective, promising mechanism [6], [7], [8], [9], [27], [28], [29], [30].

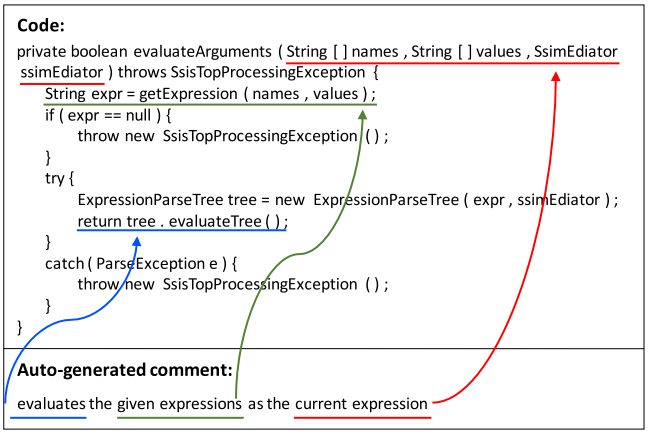


Fig. 2. A motivating example.

However, NCG is not yet a mature technique in practical scenarios due to its bad performance (i.e., the quality of the auto-generated comments) [8], [9], [31]. Meanwhile, existing NCG approaches also suffer from poor *usability*: Developers cannot easily understand and correct the auto-generated comments. Fig. 2 presents a Java code segment and its comment generated by `attendgru` [8], a typical NCG approach. The code is actually for evaluating an expression with the given arguments. But the generated comment fails to precisely present its functionality, where the "arguments" is wrongly identified as an "expression". By reading such comments, a developer cannot easily comprehend the intention of the code without carefully examining the entire code. We can see that even for such a state-of-the-art approach as `attendgru`, it is still way before it can effectively reduce human efforts in comprehending code and writing comments.

To mitigate the gap in NCG from research to practice, this work aims to address the usability issue of NCG. Actually, software usability has long been suggested as a key dimension to evaluate the quality of a software [32], [33], [34], as it significantly affects the usage of software functionalities [35]. In particular, we suggest that *NCG can be made more usable by improving its interpretability*, i.e., providing a way for developers to comprehend and improve the auto-generated comments more easily.

Our idea is inspired by the success of natural language translation (e.g., Google Translate [19]), where an explainable result is critical to its usability. In such application scenarios, two phrases in the source and target sequences typically have a natural, straightforward connection, e.g., "ingeniería de software" in Spanish and "software engineering" in English. Users can compare the resulting phrases with their counterparts in the source sequence, and examine the correctness of the translation. They can accordingly modify the result to make it more accurate.

Unfortunately, unlike natural language, the link between code and its comment is less obvious. Again, consider the example in Fig. 2. It is hard for developers to instantly know how the comment is generated with the code. They then cannot easily modify the imperfect result. Instead, they may tend to simply discard this comment and write a new one by reading the entire code (This is also confirmed in our field study, elaborated in Section 5).

But, suppose how the comments are generated with the code can be explained. As illustrated in Fig. 2, "evaluates" is from the statement that evaluates the expression; "given expressions" is from that creates the expression; and "current expression" is from the parameter list. Developers can identify whether the comments are correctly-written more conveniently. This can also allow her to improve the comments accordingly. For example, change the "as the current expression" to "with the given arguments" to describe the parameter list.

However, it is extremely challenging to provide such interpretations. It is notoriously hard to interpret how and why a deep learning algorithm produces a specific result [20]. Despite many attempts in the literature, it remains an open problem [36], [37], [38], [39], [40], [41]. Fortunately, in this task, we only need to find out which part of the code that contributes to the resulting comment words, without exploring the complex mechanisms of NMT. In other words, we can consider the NMT algorithm as a black box, and infer the connection between the code and the resulting comments. This allows us to greatly simplify the interpretation mechanism. Next, we will elaborate on how we specifically design such a framework, namely, CCLink (Code-Comment Links), to interpret the results of NCG approaches, with an aim to improve their usability.

3 CCLink Design and Implementation

As we have discussed, it is helpful to provide developers with the critical subsequences of their concern in the resulting comment, together with their corresponding code segments that contribute to the generation of such subsequences. In our following discussions, we call such subsequences *focuses*. If every *focus* can correctly describe its corresponding code segments, the comment can be acceptable to developers. Otherwise, they can accordingly modify the comment by correcting the wrong *focuses*. In this regard, CCLink finds such a mapping between each *focus* and the specific parts of code that contribute to its generation. Fig. 3 overviews our CCLink design.

Let us consider an NCG approach, the code and its resulting comment generated by the approach. CCLink first determines the *focuses* in the comment. Next, CCLink considers the NMT model in the NCG approach as a *black box* in analyzing the mapping of code segments to the *focuses*. CCLink interpret the comments in a black-box manner because we aim at providing a generic interpretation framework to improve the usability of NCG. We intentionally design CCLink to be model-independent, which relies on no knowledge of the NMT model including any white-box information of the model. Our purpose is to cope with all neural comment generation approaches, embracing emerging ones.

To this end, CCLink first generates a series of code mutants, where a mutant is a slight modification of the original code. By taking each code mutant as an input for the NMT model, CCLink obtains its corresponding comment. Then, according to the resulting comments, for each *focus*, CCLink can group the mutants into two categories: One includes those that produce the *focus*; The other contains those that do not. These two resulting categories can allow

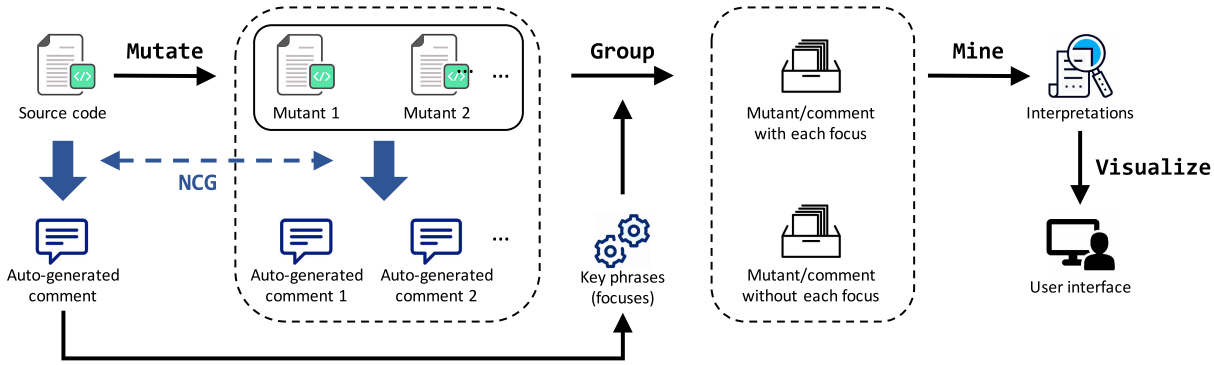


Fig. 3. CCLink overview. The workflow consists of three phases: 1) Generate code mutants and obtain corresponding comments by the NCG approach; 2) Adopt data mining approaches to interpret which parts of the code contribute to the generation of each *focus*; 3) Visualize the interpretation results to developers.

CCLink to determine which parts of the code contribute to the generation of each *focus* with data mining algorithms. Finally, such a mapping can be visualized to developers. They can examine whether every *focus* describes its corresponding code segments. This can facilitate their further correction of the comment generated with the original code.

Next, we will first discuss how CCLink finds the *focuses*, generates code mutants, and obtains the resulting two categories of data for each *focus* in Section 3.1. The interpretation mechanisms of CCLink will be discussed in Section 3.2. Finally, we will elaborate CCLink's user interface in Section 3.3.

3.1 Data Preparation

We first describe how CCLink determines the *focuses* in the comment. Note that comments are natural language sentences. There is already a rich body of literature in the natural language processing community that studies how to find the key information in a natural language sentence (e.g., [42], [43], [44]). This allows us to conveniently borrow the existing, well-accepted concepts and approaches to retrieve the *focuses*. Hence, CCLink conducts this task by invoking a plugin algorithm. Such a plugin can easily be substituted with any other similar algorithms. In particular, in our study we use RAKE [45], a widely-used rapid automatic keyword extraction algorithm to retrieve the *focuses*. It can identify important phrases, i.e., those conveying key information, in a sentence.

With a set of *focuses*, CCLink's task now is to find the code segments that contribute to the generation of the *focuses*. An instant idea is to analyze how the NMT model processes the input code, and accordingly determine which parts of the code contribute the most to the production of each *focus*. However, interpreting a deep learning model like NMT is still a challenging, open problem [36], [37], [38], [39], [40], [41]. Moreover, as we have discussed, CCLink aims at providing a generic interpretation framework to improve the usability of NCG. Such an interpretation attempt should be orthogonal to the line of research efforts that improve the NMT model *per se*. Hence, CCLink should be an NMT model-independent approach, which cannot base its design on analyzing a specific NMT model. With these considerations, CCLink resorts to a *black-box* approach to analyze which parts of the code contribute to the generation of each *focus*.

To this end, CCLink employs a code mutation-based method: It removes only a small segment from the original code to generate a code mutant. The purpose is to facilitate the examination of whether the removed code segment contributes to the generation of a *focus*. It also carefully avoids changing the code syntax structure, since some NCG approaches (e.g., [7], [29]) may require such structure information in generating comments.

Now we discuss how CCLink performs such removal to generate code mutants. First, CCLink should determine the granularity of the code segment to be removed. An instant measure is to remove some lines of codes. But, such a granularity is not suitable, since the original code *per se* may typically have only a few lines (We will provide our experimental study on such a granularity in Section 4). Consider the fact that in practical software projects, the intention of code is typically suggested by the natural language words in the code (e.g., the variable names) [21], [22]. CCLink allows developers to obtain a fine-grained link between the code and the *focuses* by removing such words in the original code.

Specifically, CCLink conducts tokenization on the original code as follows. It first considers only the English alphabet and disregards other symbols by substituting them with spaces. For example, the "if(isInitialized)" will be transformed to "if isInitialized". In this way, the snake-case names can also be separated into words, e.g., "file_name" will be transformed to "file name". Then CCLink removes the programming-language keywords (e.g., "public" and "int") except those for flow control (e.g., "if"), since the flow-related keywords may determine the behavior of the code. CCLink thus obtains a sequence of words (not necessarily natural language ones). It then tries to split each word if they are not in English vocabulary. This is mainly for separating camel-case names. For example, "fileName" will be transformed into "file name". Eventually, CCLink transforms the code into a set of tokens (i.e., words). It then randomly selects the tokens from the list, and removes them from the code to generate code mutants. A mutant m can thus be modeled by the set of tokens T_m removed from the original code. Fig. 4 shows the above workflow of generating a code mutant. The removed tokens will be replaced with blank placeholders to keep the code structure unchanged.

Finally, CCLink uses the target NCG approach to generate comments based on these code mutants. In this way, for

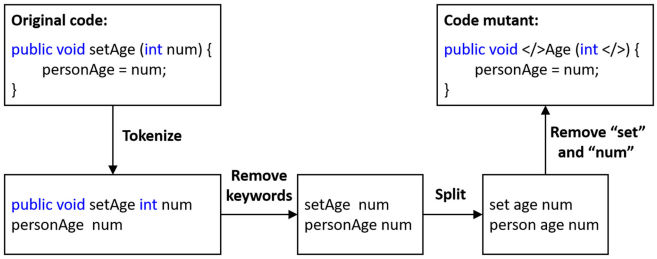


Fig. 4. The workflow of generating a code mutant.

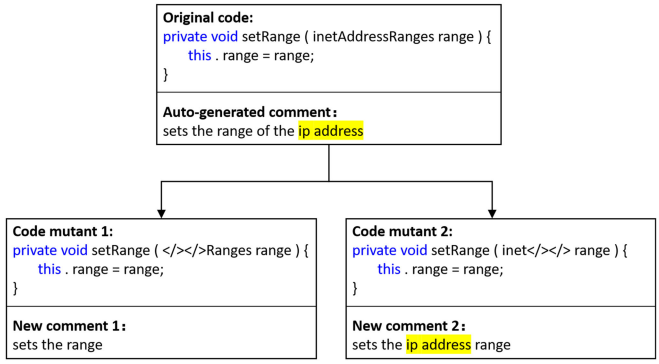


Fig. 5. Example of different impacts caused by removing the same token.

each *focus* k, these code mutants can be divided into two categories: one can produce the comments with the k (denoted by M_k) and the other without (denoted by $\overline{M}_k$).

3.2 Comment Interpretation With Contribution Mining

After obtaining the code mutant categories, we can preliminarily infer the impact of the removed tokens on a specific *focus*, i.e., if the removal of some tokens results in the absence of *focus* k (The corresponding code mutants are in $\overline{M}_k$), it is reasonable to believe that these tokens have more impact on k. However, infering what exactly causes the generation of each *focus* is not easy. We can find in our experiments that the absence/presence of a specific token does not always result in the absence/presence of a *focus*.

For example, consider the simple code shown in Fig. 5 and `attendgru` [8] as our NCG approach. A mutant where the tokens "inet" and "address" are both removed from the original code cannot produce a comment with the *focus*"ip address". However, a mutant where the tokens "address" and "ranges" are both removed can produce the *focus*"ip address". We can see that the absence of the token "address" in the mutants may not always indicate the absence of a *focus*"ip address" in the generated comments.

This is actually not surprising due to the complication of deep learning. An NMT model is non-linear in nature. Its behaviors (i.e., whether to produce a *focus*) cannot be simply determined by checking whether a specific token exists or not. In this regard, CCLink models the comment interpretation problem as one that infers a combination of tokens, which are more likely to cause the NMT model in producing the *focus* of concern.

We consider token combinations because it is more suitable to model the non-linear nature of an NMT. Moreover, presenting several tokens to the developer can provide more information for her to determine the correctness of the *focus*, than presenting only one single token. Inferring such a token combination is essential to provide an *approximate* black-box model of the non-linear, complicated behaviors of the target NMT. Hence, we have to resort to heuristic methods. Next, we customize three interpretation methods, namely, FreqCM, LimeCM, and AnchorCM as examples.

3.2.1 FreqCM

The first method FreqCM (Frequent pattern-based Contribution Mining) conducts token combination inference with frequent pattern mining [46]. Specifically, given the two sets of mutants M_k and $\overline{M}_k$, consider a token combination that exists frequently in the collection of $T_{m'}$ $(\forall m' \in \overline{M}_k)$, but exists rarely in the collection of T_m $(\forall m \in M_k)$. We can infer that the code segments containing such a token combination are removed from the original code, the *focus* k is more likely to be absent in the resulting comments. Thus, such a token combination can be considered as that contributing the most to the generation of the *focus*.

Hence, FreqCM applies a similar procedure as that in the Apriori algorithm, a classic frequent pattern mining algorithm [47]. Given a token combination tc, the collections of $T_{m'}$ $(\forall m' \in \overline{M}_k)$ and T_m $(\forall m \in M_k)$, it relies on a breadth-first search strategy to count the numbers of occurrences of tc in both collections respectively. The occurrence numbers are divided by their corresponding collection sizes for normalization purpose, the values of which are denoted by sup'_{tc} and sup_{tc} respectively. $sup'_{tc} - sup_{tc}$ is then deemed as the support of the combination. A higher support indicates the combination occurs more frequently in the collection of $T_{m'}$ $(\forall m' \in \overline{M}_k)$ than it occurs in the collection of T_m $(\forall m \in M_k)$. The support is considered as the *contribution* of the token combination. We formalize this process as follows:

$$sup'_{tc} = \frac{count'(tc)}{|T_{m'}|}$$
$$sup_{tc} = \frac{count(tc)}{|T_m|}$$
$$con^k_{tc} = sup'_{tc} - sup_{tc}$$

where $|T_{m'}|$ and $|T_m|$ represent the collection sizes of $T_{m'}$ and T_m, $count'(tc)$ and $count(tc)$ represent the occurrence numbers of tc in $T_{m'}$ and T_m respectively, con^k_{tc} is the contribution value of tc on *focus* k.

FreqCM then recursively extends such subsets until no further successful extensions are found. In this way, it can obtain the token combination with the maximum contribution, and consider it as the result. In case the two token combinations have the same highest contribution value, FreqCM chooses the one with more tokens, in order to provide more information to developers.

3.2.2 LimeCM and AnchorCM

Note that FreqCM is not the only choice. There is also much recent work that attempts to interpret sequence-input deep learning models. Two representative model-independent interpretation approaches are LIME[17] and Anchor [18]. An instant consideration is that CCLink may resort to such existing approaches.

However, recent approaches (e.g., LIME and Anchor) are generally designed to interpret deep learning models that perform classification tasks, instead of NMT. Fortunately, we show that our interpretation problem can also be transformed into a set of classification interpretation problems, where existing approaches can be applied. In particular, consider the mutants as the inputs, and whether or not a *focus* is present in their corresponding generated comments as a Boolean output. The NMT model can then be deemed as one that conducts a two-class (i.e., presence/absence) classification task for the *focus*. In this way, CCLink can apply both LIME and Anchor to determine the contributions of the token combinations. Specifically, we name them LimeCM (LIME-based Contribution Mining) and AnchorCM (Anchor-based Contribution Mining). Next, we will first briefly introduce LIME and Anchor, followed by showing how we tailor LimeCM and AnchorCM for our comment interpretation task.

LIME [17] constructs a linear, interpretable model to locally fit the behavior of a complicated deep learning model. In particular, LIME randomly perturbs an original input sequence by randomly removing its elements. It takes these perturbed sequences as inputs and applies the target deep learning model to obtain their corresponding classification results. The sequences, together with the results, are used as training cases to train a simple, linear classifier. Based on the parameters of the linear classifier (i.e., the weights of the elements in the sequences), LIME can model the contributions of the elements. Such contributions are then considered as an approximation to interpret how each element in the input sequence influences the result of the deep learning classifier.

Anchor [18], in contrast, finds the invariant *if-then* rules. Specifically, such an *if-then* rule indicates that *if* an input contains a set of specific subsequences, namely *anchors*, *then* the model will produce a specific classification result. It perturbs the input sequence with a natural-languages word embedding-based method, and lets the model produce the results. It then accordingly calculates a contribution value of the subsets of the element with the KL-LUCB algorithm [48]. The ones with high contributions are considered the anchors.

LIME perturbs the input sequence by randomly removing its elements. Anchor perturbs the input sequence by substituting similar natural-language words. But their perturbation methods are orthogonal to their inference mechanisms. Hence, in LimeCM and AnchorCM, we substitute the perturbation methods with our code-mutation method discussed in Section 3.1 as it is more suitable for programs.

Since LIME calculates the contribution of each token separately, we consider the top N tokens with the highest contributions as the resulting token combination. The contribution of token combination is considered as the average of the N values. N is a parameter in LIME with default value six [17]. LimeCM adopts this default setting. Similarly, the contribution value of tc on k is denoted as con^k_{tc}, and its formal definition is as follows:

$$con^k_t = weight(t)$$
$$con^k_{tc} = \frac{\sum con^k_t(\forall t \in T_{top_N})}{N}$$

where $weight(t)$ means the weight assigned to token t in the liner model, T_{top_N} is the collection of tokens with top N contribution value con^k_t.

In contrast, Anchor *per se* finds token combinations. So, AnchorCM applies Anchor's inference mechanism, and takes the anchor with the highest contribution value (also denoted as con^k_{tc} for consistency) as the result.

The computational complexity of LIME is linearly related to the number of training samples, i.e., the number of mutants. Its efficiency is acceptable. Anchor, in contrast, resorts to an exhaustive search of token combinations, which may result in low computational efficiency. We will study their computational efficiency in Section 4.

Finally, note that our comment interpretation methods FreqCM, LimeCM, and AnchorCM may be further substituted with one that customizies other sequence-input deep learning interpretation approaches, including the possible improved approaches in the future. Our AnchorCM and LimeCM can serve as examples to guide such customization. In this regard, CCLink again resorts to a plugin-based design, where new deep learning interpretation approaches can be easily applied.

3.3 Visualization and User-Interaction Considerations

Our comment interpretation aims to facilitate developers to comprehend code and write comments. It is important to provide a usable graphic user interface module for CCLink.

First, CCLink should present an easy-to-comprehend illustration of the code-comment links. Inspired by Tarantula [49], a classic visualization tool for fault localization, CCLink also displays *focus* and its corresponding token combination in a spectrum-alike manner. We color each *focus* and its corresponding parts in the source code the same, to facilitate a clear understanding of the link between them.

In addition, the execution time of tool during UI interaction with users is a critical concern to its practical application, as confirmed by tremendous HCI research. To reduce such time, we divide the execution of NCG and the CCLink's interpretation process into two parts: the light-weighted UI and the computation-intensive backend running with the target NCG approach. Such a design also allows the UI of CCLink to be implemented as an IDE plugin easily.

The backend implements the mechanism discussed in Sections 3.1 and 3.2. Since in our use scenario, the target code to be comprehended is available beforehand. This allows us to conduct the NCG process (which may require over 100ms to complete in desktop computers [7], [8]) and the comment interpretation at the backend offline. In other words, it is conducted not during the developers are comprehending the code and modifying the auto-comments, but beforehand. Its results are then saved.

When a developer intends to comprehend the code and modify the auto-comments, the UI of CCLink loads the results and present to the developers the graphic illustration. Such illustration of results is light-weighted, which do not incur noticeable latency to the developers, since the results are already available.

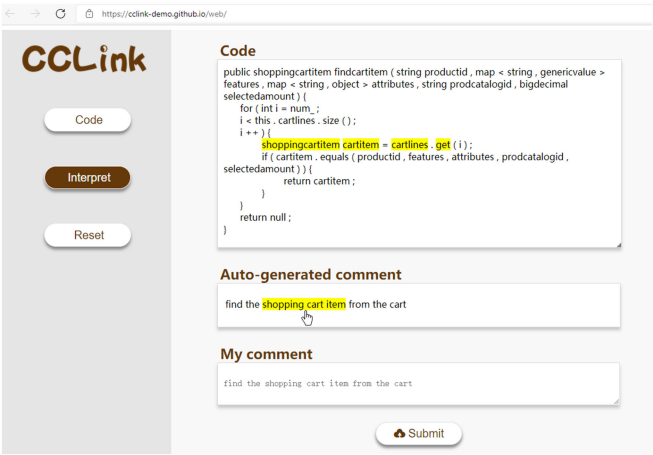


Fig. 6. User interface of CCLink prototype. It can also be found at https://cclink-demo.github.io/web/.

Fig. 6 presents the user interface (UI) of our CCLink prototyping implementation. The UI includes three parts: The code, the auto-generated comments by an NCG approach, and an edit box for developers to correct the comments. A developer can load the code of concern, then generate its comment with the NCG approach. CCLink then interprets the results. When she moves her mouse pointer to the *focus* of her concern, the interpretation will be shown by coloring the *focus* and the corresponding parts in the source code. In this way, the developer can check whether the *focuses* correctly describe the intention of the code. If the answer is *no*, she can accordingly modify the comment. We will present an empirical study on the usability of such a comment interpretation tool, which will be presented in Section 5.

4 Performance Study

In this section, we present our performance study on CCLink. In particular, we focus on the following research questions:

- *RQ1:* What is the interpretation quality of CCLink with different comment interpretation approaches?
- *RQ2:* What is the computational efficiency of CCLink with different comment interpretation approaches?
- *RQ3:* Whether CCLink is model-independent, i.e., extendable to interpret different NMT models?

4.1 Experiment Settings

NMT Models. We consider `attendgru` as our target NMT model [10] when answering RQ1 and RQ2, because it is a representative, state-of-the-art approach, based on an advanced deep learning technique, i.e., the *Attentional RNN Encoder-Decoder* neural network. In answering RQ3, we also test CCLink on another NMT model, namely `ast-attendgru` [8]. It considers two types of information on source code, i.e., the code text, and its syntax tree.

We build the original neural networks of `attendgru` and `ast-attendgru` respectively with the *Keras* (version 2.2.4) deep learning framework, which employs *TensorFlow 1.12* to conduct the backend deep learning tasks. We use the default settings proposed by the original model designers [8], [10]. The code for modeling training can be found at https://github.com/mcmillco/funcom [8]. Regarding the training arguments, we set *batch_size* to 200 and *epochs* to 100. We use the default values for the rest of the arguments.

TABLE 1
Dataset Statistics

	∗-20		∗-100	
	Code	Comment	Code	Comment
OrigTrain	13.1	7.6	47.7	7.9
OrigTest	12.3	7.8	49.8	8.1
DeepCom	11.3	8.1	51.0	8.0

Code size is in number of tokens; Comment size is in word counts.

Datasets. We consider two real-world Java method corpora to build our datasets. One is the corpus of `attendgru` [8], the other is that of DeepCom [7]. They include both Java methods and their comments that describe their functionality.

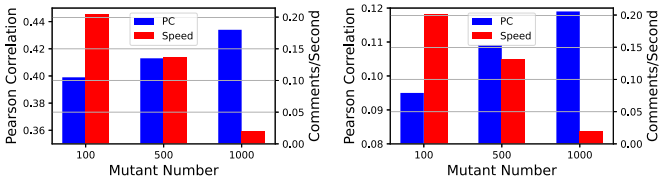
For the first corpus, we first select samples from its training set. In this way, we randomly select 300 methods with small sizes ($\leq$ 20 tokens) and 300 methods with larger sizes (21-100 tokens), in order to study the influence of input size on CCLink performance. They are called OrigTrain-20 and OrigTrain-100, respectively. Considering that CCLink is not designed for producing comments, instead, it is for interpreting the comments. We select samples from the training set with an aim to produce more accurate results. This allows us to examine more easily whether CCLink can precisely interpret the generated comments.

With a similar procedure, we also randomly select samples from the test set of the `attendgru` corpus, and from the DeepCom corpus. We form OrigTest-20, OrigTest-100, DeepCom-20, DeepCom-100 datasets, respectively, where 20 and 100 have the same meaning as in OrigTrain-20 and OrigTrain-100. The number of tokens is limited to 100 due to the limitation of the input size of `attendgru`. Table 1 illustrates the average sizes of the Java methods and their comments generated with `attendgru`, for our six datasets.

Setting of Mutants. The number of code mutants directly impacts execution efficiency and interpretation quality of CCLink. With more mutants, the interpretation quality becomes higher because more token combinations are explored. However, the execution time also increases due to the increased computation overhead. To show the effect of the different number of mutants, we evaluate CCLink (with FreqCM) on small-scale data [1] with three different mutant settings, i.e., 100, 500, and 1000. Fig. 7 shows that CCLink achieves a good trade-off between interpretation quality (how to obtain the quality measure will be elaborated in Section 4.2) and efficiency with around 500 mutants. Therefore, our following discussions only report the results where the number of code mutants in each run is fixed to 500.

Experiment Environment. All experiments are running on a server with a 6-Core Intel i7-6800K 3.40GHz Processor and 48GB DDR4 memory. Deep learning approaches are boosted with an Nvidia GeForce GTX 1070 GPU. The server is running over 64-bit Ubuntu 16.04.1 with Linux kernel 4.15.0.

1. We randomly sample one-third of the data from OrigTest-20 and OrigTest-100.



(a) Results on OrigTest-20 (b) Results on OrigTest-100

Fig. 7. The effect of different number of mutants on interpretation quality (measured by Pearson correlation, discussed in Section 4.2) and execution efficiency (measured by computation throughput, i.e., the number of comments processed per second).

4.2 RQ1: Interpretation Quality

The key for CCLink to improve the usability of NCG is the interpretation mechanism. Therefore, we should evaluate the interpretation quality of CCLink, i.e., how a focus explains its corresponding focused code. However, this is not a straight-forward task. Actually, the quality is a subjective consideration of the developer who reads and modifies the comments. Unfortunately, there is generally no ground-truth value of whether a code-comment link is correct. Hence, to compare different interpretation methods, we resort to a heuristic following the standard method widely adopted by previous studies [50], [51]. We consider that if an interpretation method is more reasonable, the contribution values of the resulting interpretation (the code-comment links) should tend to be more semantically correlated with the text similarity between the resulting focused code and the interpreted focus. Such a natural-language perspective can reflect the developer's perception on the natural-language texts (i.e., the words in comments or codes) [11], and thus indicate the interpretation quality.

In this regard, similarly as the previous work [50], [51], we also adopt Pearson correlation (ranging from -1 to 1) [52] to measure such correlation, and deem it as the result quality of an interpretation method. Specifically, the Pearson correlation is calculated on the tokens determined to be important by CCLink and word embeddings produced by GloVe [53], a widely-accepted word embedding model. Formally, the Pearson correlation $\rho_{X,Y}$ between two sets of data X and Y is calculated as:

$$\rho_{X,Y} = \frac{cov(X,Y)}{\sigma_X \sigma_Y} = \frac{E(XY) - E(X)E(Y)}{\sqrt{E(X^2) - E^2(X)}\sqrt{E(Y^2) - E^2(Y)}}$$

where $cov(X,Y)$ is the covariance of X and Y, σ_X and σ_Y are their standard deviations, and $E(\cdot)$ measures a mean. In our task, X denotes the contribution values of the resulting token combinations calculated by CCLink (a set of con_{tc}^{k} described in Section 3.2). Y denotes the text similarities between the token combinations and their corresponding *focuses*, calculated by GloVe [53]. The correlation between X and Y measures how semantically close the interpretations are to their corresponding *focuses*. In this way, we can examine how the interpretation methods perform. The higher the Pearson correlation, the better the interpretation method.

Since CCLink interprets auto-generated comments with three contribution mining methods we design (i.e., FreqCM, LimeCM, and AnchorCM), we compare CCLink with each interpretation method. In addition, to verify that a sophisticated contribution mining method like FreqCM, LimeCM, or AnchorCM is important in finding high-quality interpretations, we also design a greedy-algorithm-based approach, namely, CFL (Counting based on Frequent Lines), as a baseline to compare with these three methods. CFL finds the lines of code that may contribute the most to the generation of a *focus*. It removes some lines of code to generate code mutants. Similarly, the comments of the mutants are generated with the target NCG approach, each at a time. Then for each code line, CFL counts the number of times that its absence causes the absence of the *focus*. CFL considers the lines with the maximum number as that contribute to the generation of the *focus*.

TABLE 2
Pearson Correlation Values When Using CFL and CCLink With Different Interpretation Methods (FreqCM, LimeCM, and AnchorCM)

	OrigTrain-20	OrigTest-20	DeepCom-20
CFL	0.004	0.079	-0.098
FreqCM	**0.359**	**0.396**	**0.256**
LimeCM	0.140	0.199	-0.002
AnchorCM	-0.291	-0.268	-0.318
	OrigTrain-100	OrigTest-100	DeepCom-100
CFL	-0.108	-0.140	-0.063
FreqCM	0.210	0.199	**0.083**
LimeCM	**0.276**	**0.282**	-0.122

Table 2 illustrates the Pearson correlation values calculated on our test data when using CFL and CCLink with different interpretation methods (i.e., FreqCM, LimeCM, and AnchorCM) [2]. We can see that when applying FreqCM and LimeCM as interpretation methods, CCLink perform much better than CFL. In particular, many of the results on CFL are negative, indicating that the token combinations are less semantically-relevant to the *focuses*. Such resulting interpretations may, on the contrary, cause confusion to developers. The results indicate that it is important for CCLink to include a more sophisticated comment interpretation method (e.g., FreqCM and LimeCM), rather than just using simple, straight-forward solutions like CFL.

We also find that the results on OrigTrain-* are close to the results on OrigTest-* and DeepCom-*. Although the comments in OrigTrain-* are more accurate (since they are from the training set of `attendgru`), this does not significantly affect the quality of the interpretations produced by CCLink. Moreover, we can see that the performance of AnchorCM is not very satisfactory, showing that some sophisticated interpretation mechanisms do not always perform well. It is still an open problem to tailor better comment interpretation mechanisms for CCLink.

4.3 RQ2: Computational Efficiency

We evaluate the computational efficiency of CCLink to examine whether it is feasible in practice. Hence, we measure the average execution time of CCLink with different

2. As CCLink with AnchorCM cannot produce a result less than ten minutes for each method in the datasets with larger code size (i.e., OrigTrain-100, OrigTest-100, and DeepCom-100), we do not apply AnchorCM to those datasets.

TABLE 3
Average Execution Time (in Second) of CCLink With Different Interpretation Methods on Each Java Method

	OrigTrain-20	OrigTest-20	DeepCom-20
FreqCM	**8.01**	**7.69**	**7.86**
LimeCM	9.48	9.35	10.02
AnchorCM	60.97	51.76	60.98
	OrigTrain-100	OrigTest-100	DeepCom-100
FreqCM	**7.08**	**7.75**	**7.43**
LimeCM	9.20	10.65	10.64

TABLE 4
Pearson Correlation Values and Average Execution Time (in Second) on Each Java Method (on `ast-attendgru`)

(a) Pearson correlation values				
	OrigTrain-20	OrigTrain-20	OrigTest-100	OrigTest-100
FreqCM	0.344	0.422	0.186	0.180
LimeCM	0.170	0.183	0.267	0.155
(b) Average execution time				
	OrigTrain-20	OrigTrain-20	OrigTest-100	OrigTest-100
FreqCM	7.36	7.45	8.58	7.73
LimeCM	8.89	10.53	9.41	11.16

interpretation methods on each of the Java methods in the six datasets. The results are shown in Table 3.

We can see that CCLink with AnchorCM has the largest execution time, compared with the other two. It takes over 50 seconds on average to complete a method in the datasets with small code size (i.e., OrigTrain-20, OrigTest-20, and DeepCom-20). It even fails to produce results in 10 minutes for each method in the rest three datasets. In such a long time, a developer can even read/comprehend the code by herself. This indicates that AnchorCM, although shown as a sophisticated interpretation method, does not show advantages in our task.

The average execution time of CCLink with FreqCM to process a method is about 7 to 8 seconds, which is slightly faster than CCLink with LimeCM. We also find that their execution time on methods with different sizes (those in the $*$-20 datasets and those in the $*$-100 datasets) is close. This is because the target NMT, i.e. `attendgru`, will always pad its inputs to 100 tokens for all 500 mutants of a method. The time to generate comments of the mutants is hence similar. Moreover, the execution time of FreqCM and that of LimeCM are both linearly related to the number of mutants. Therefore, the execution time of CCLink with FreqCM/LimeCM will not vary significantly given methods with different sizes.

In conclusion, the computational efficiency of CCLink with different interpretation methods varies. CCLink with FreqCM, which requires less than 10 seconds to complete a method, is a good choice. Anchor, although shown as an advanced interpretation approach for deep learning, does not show advantages in our problem setting.

4.4 RQ3: Model Extendablity

As introduced in Section 3, our design of CCLink does not rely on the specifics of the target NMT model. It is a model-agnostic approach. We verify this by applying CCLink to interpreting comments generated by different NMT models. Specifically, we test CCLink on `ast-attendgru` [8], another NMT-based comment generation approach. It uses both code text and th corresponding abstract syntax tree as the model inputs.

Since CCLink with AnchorCM suffers bad efficiency, in this study, we only consider CCLink with FreqCM and with LimeCM. Similarly, we evaluate the Pearson correlation and the execution time of CCLink on the same datasets except DeepCom-$*$, because they cannot reproduce the required AST. Table 4 shows the results when the target NCG is `ast-attendgru`. We can see that CCLink performs similarly in terms of both Pearson correlation and execution time. This proves that CCLink is model-agnostic and can be applied to interpret different NCG approaches.

5 Empirical Study

CCLink interprets the comments generated by NCG approaches, with an aim to facilitate developers to comprehend code and correct the comments. Hence, it is critical to evaluate CCLink from the perspective of developers. To this end, we conduct an empirical study to verify the usability of CCLink. Our study aims to answer the following research questions:

- *RQ4:* Can CCLink improve the efficiency of developers to comprehend and correct auto-generated comments?
- *RQ5:* Can CCLink improve the quality of comments corrected by developers?
- *RQ6:* What impact does CCLink have on developers' workflow when using NCG tools?
- *RQ7:* How satisfied are developers with CCLink?

5.1 Experiment Settings

Our study consists of two stages. First, we design a comment correction task to simulate code comprehension in practice. Then, we communicate with developers to understand their behaviors and thoughts. The settings for each stage are illustrated separately.

5.1.1 The First Stage

In the first stage, we examine whether CCLink can help developers in code comprehension and comment correction. We recruit participants with over five-year Java programming experience based on our social connection, and then extend the participant set with snowball sampling [54]. In total, we recruit 32 qualified participants, including software engineers in global software companies and graduate students majoring in computer science. We consider them as representative target users of CCLink.

We ask each participant to conduct an online experiment by accessing the website of CCLink, which typically lasts for 40 minutes. This experiment simulates the scenario where developers intend to contribute comments to open source projects using NCG tools. In this scenario, developers read code written by others. Hence, we adopt Java code from existing projects. We randomly select 20 Java methods

from DeepCom-100 dataset (described in Section 4.1) for each of our participants. Similar to the setting of the performance study, we generate the comments with `attendgru`.

In the experiment, each participant is asked to perform the following two tasks with CCLink. *Task 1*: Read each of the 10 Java methods and its auto-generated comment, then correct the comment to best describe the method, *without* enabling the interpretation function of CCLink. *Task 2*: Read each method in another 10 Java methods and its auto-generated comment, and correct each comment to best describe the method, *with* the comment interpretations produced by FreqCM.[3]

Finally, we extract the comments corrected by the participants and the time they spent reading and correcting each comment. We also take video-record of the entire experiment. Our purpose is to guarantee the reliability of data (e.g., a participant is not interfered by another task), and to facilitate further behavior analysis.

5.1.2 The Second Stage

Based on the above experiment, we further explore the reasons for the efficiency differences of the developers in completing the two tasks. We investigate the impact of CCLink on developers' workflow. To this end, we organize a *think-aloud workshop* [55], where the participants are required to tell us how they work. We invite seven participants to the workshop, identified as P1 to P7.

During the workshop, with the same setting in stage one, we present each participant with one randomly selected Java method, an auto-generated comment for that method, and the interpretation given by CCLink. Each participant is required to reproduce the practice in the first stage, and at the same time report her behaviors and thoughts using *think aloud* approach [55].

To better understand their workflow, we record and analyze all the screen activities of the participants during the workshop after acquiring their consent. To ensure data validity, we also conduct a *round table discussion* with all the 7 participants in this stage, discussing their comment writing practice with and without CCLink during the workshop.

We also conduct semi-structured interviews with the participants. Our focus is their user experiences. We follow the wide-adopted semi-structured interview methodology: We conduct our interview without "reading" the questions of our interests. Instead, we adapt our questions to our communication contexts. For example. We ask our participants "when you modify unsatisfactory comments, do you read the interpretations?". If the answer is *yes*, we will further ask "do you find the interpretations make any sense or most of time, they are misleading." If our user's response is that the interpretations are accurate, we will ask whether the interpretations help in modifying the comments and ask them to elaborate how they conduct their tasks accordingly. Otherwise, we may ask the user to give examples and talk more on how they feel and how they continue their tasks when they find the interpretations inaccurate. We aim to interview in a manner that our users tell more on their stories (how they conduct the tasks and how they feel under different situations), instead of asking them to answer a set of predefined *yes/no/why* questions. We aim to obtain more accurate understanding on our participants with such communications.

- After providing the interpretations of CCLink, do you think the efficiency of modifying comments has improved and why?
- Please rate your satisfaction level (ranges from 1 to 10) with the comments you modified in Task 1 and Task 2.
- Can CCLink provide helpful links between focus phrases in comments and focus code?
- In what aspects do you think CCLink helps you comprehend and modify the comments?
- If you consider CCLink is not helpful, please describe the reasons.
- Are you willing to use NCG tools in practice *without* interpretation? Will your willingness to use NCG tools improve *with* CCLink?

Fig. 8. Key questions in the questionnaire.

Finally, in order to investigate the developers' satisfaction with CCLink comprehensively, we send a questionnaire to all the 32 participants to ask about their opinions on CCLink. The questionnaire is designed according to our interview experiences. The questionnaire include 22 questions, including general questions regarding their overall experiences and specific ones including how they evaluate the links in improving their workflow. We summarize the questions in the questionnaire into several key ones presented in Fig. 8, where we merge related questions to save space. To facilitate further research, we also release the full questionnaire at https://github.com/CCLink-demo. We will report our findings based on both the results of the questionnaire and the workshop (including the round table discussion and the interviews) in what follows.

5.2 RQ4: Efficiency Improvement

We first analyze whether the interpretations improve the efficiency of our participants to correct the comments. Fig. 9 shows the average time for each participant to correct comment of each Java method in the two tasks.

We can observe that the time for most participants in Task 2 (with interpretation) is shorter than in Task 1 (without interpretation). Averagely, our participants require 128.8 seconds to correct the comment of each method in Task 1. The time is reduced to 93.6 seconds in Task 2. The results indicate that interpretations given by CCLink indeed save the time of developers to comprehend and modify the comments in practice.

We further conduct a significance test to explore whether there is a significant difference in the time spent by the participants on the two tasks. Specifically, we adopt *Paired t-Test*, which is suitable for testing two sets of paired samples intervened by a certain indicator. In our experiment, the time spent by each participant in the two tasks is considered as a paired sample. We first make a hypothesis (denoted as H_0) that there is *no* significant difference in the average time spent by the participants in the two tasks.

3. We choose FreqCM as our comment interpretation method as it demonstrates good performance and computational efficiency, which has been reported in Section 4.

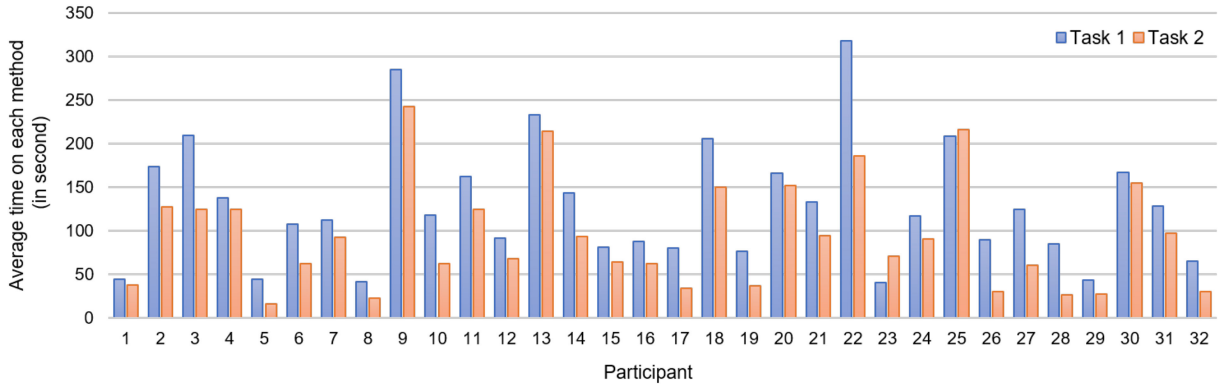


Fig. 9. Average time for each participant to correct comment of each Java method in the two tasks.

Then we calculate the t value based on the 32 paired samples, and determine to accept or reject H_0 based on the t value. Table 5 shows the result of t-Test. In our experiment, the degree of freedom df is 31 (one less than the sample size), and we set the significance level α as 0.01. In this case, we can reject H_0 if the t value is greater than 2.744 (the calculated value is 6.949). The result indicates that the time spent by the participants in Task 2 is significantly shorter than in Task 1.

5.3 RQ5: Comment Quality Improvement

Besides task-completing efficiency, it is also important to study whether CCLink improves the quality of comments corrected by the participants. To answer this question, we calculate the BLEU scores and edit distances (i.e., Levenshtein distances) between user-modified comments and the ground truth, i.e., the human-written comments provided in the dataset, in the two tasks. We use these two metrics as comments are human-readable natural languages. The similarity between natural language texts are generally measured by the two metrics [12]. The BLEU scores and edit distances are calculated on word level. A higher BLEU score and a smaller edit distance represent a better result. We obtain the average values of the BLEU scores and edit distances of all our partipants' result. We find that in Task 2, the user-modified comments can achieve both a higher BLEU score (improved by 0.11 compared with Task 1 averagely) and a smaller edit distance (improved by 1.27 compared with Task 1 averagely) with the ground truth. Again, we also conduct Paired t-Test on these results. The test setup is the same as described in Section 5.2. Table 6 shows the results of the t-Test. The results indicate that with the interpretation of CCLink, the quality of comments corrected by the participants has significantly improved.

In addition, the participants also confirm that CCLink improves the quality of their modified comments. Based on the results of the questionnaire, 25 participants consider the quality of their modified comments in Task 2 is higher than those in Task 1. We also ask participants to read their modified comments and rate their satisfaction level (from 1 to 10) with their modification. The results show that the average level is 5.94 in cases without interpretations. Whereas, it substantially increases to 7.47 when they use CCLink with comment interpretations. These results again indicate that the interpretability studied in our work should be of critical concern.

5.4 RQ6: Workflow Changes

Instead of being satisfied by the fact that CCLink elevates the efficiency of comment correction, we decide to take a step further and illustrate why CCLink can achieve the efficiency. We analyze the screen activities recorded during the workshop, as well as the records of the round table discussion. Based on these materials, we summarize the workflow of participants with/without CCLink, shown in Fig. 10.

We find that *without* CCLink, the participants typically complete their tasks with three steps: 1) Get a first impression of the presented method by quickly glancing at the method name and the auto-generated comment. 2) Read the whole method body to fully comprehend the meaning of the method. 3) Rewrite the comment based on their understanding of the method. We observe that step 2 typically requires considerable time, as the difficulty of understanding the method body is non-trivial. As P6 said during the discussion: "*I often track the inputs, calculating key parameters while reading code in the method body. sometimes I have to read some lines repeatedly to ensure I have understood them correctly.*"

However, when the participants conduct the tasks *with* CCLink, their workflow changes a lot. While the first step is similar to the above, in step 2 the participants do not read the whole method body. Instead, they check specific code snippets based on the code-comment link provided by CCLink, then determine the correctness of the comment words. If a participant thinks the comment is inaccurate in

TABLE 5
The Result of Paired t-Test Based on the Time Spent by the 32 Participants in the Two Tasks

Degrees of freedom (df)	Significance level (α)	t	Accept/Reject H_0
31	0.01	6.949	Reject

TABLE 6
The Results of Paired t-Test Based on BLEU Scores and Edit Distances Obtained From the 32 Participants in the Two Tasks

	df	α	t	Accept/Reject H_0
BLEU scores	31	0.01	15.122	Reject
Edit distances	31	0.01	12.758	Reject

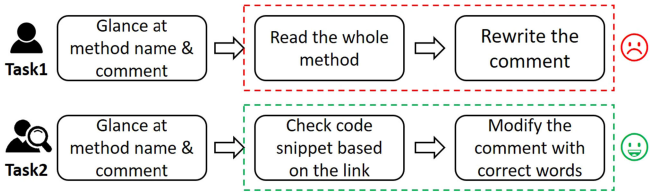


Fig. 10. Comment correction workflow comparison.

describing the method, she typically substitutes certain words with suitable ones, and leaves the sentence structure unchanged.

The result indicates that CCLink changes the original read-understand-write practice into a more effective revision-based workflow. The perceived effectiveness of CCLink in the workflow reported by the participants can be summarized as follows. 1) With the interpretations, they can focus on the specific code snippet, which saves their time by avoiding reading the entire code. 2) The interpretations, in turn, allow them to quickly examine the correctness of comments. 3) The key information highlighted by CCLink can inspire them to revise the auto-generated comments precisely, and also efficiently by allowing them to reuse the sentence structure.

5.5 RQ7: User Satisfaction

To obtain a more comprehensive view of the participants' satisfaction towards CCLink, we analyze the opinions of all the 32 participants in the questionnaire and the results obtained from the workshop.

The majority of our participants explicitly consider CCLink helpful. During the round table discussion, 71.4% (5/7) of our participants explicitly propose that CCLink helps in both program comprehension and comment modification. The effectiveness of CCLink is also confirmed in the questionnaire where 81.3% (26/32) of the participants express that the interpretations provided by CCLink substantially facilitate their tasks.

We are particularly interested in analyzing why some participants do not explicitly consider interpretations helpful. According to the feedback of six such participants in the questionnaire, one reports that when he reads the first auto-generated comment, he considers that the comment is irrelevant to the code. He feels bored and does not read any auto-generated comments from then on. The other five participants share similar considerations, although their attitude is less negative. Actually, their negative attitude is due to the bad quality of the auto-generated comment, instead of CCLink. This confirms that bad auto-generated comment incurs unwillingness to use NCG, and hence our direction in this work to improve NCG usability is important. Moreover, their negative attitude also gives us inspirations to improve CCLink. For example, in a scenario where the auto-generated comment is of poor quality, e.g., the semantic similarity between code and the comment is extremely low, CCLink could reduce the interpretations of the comment. In this way, CCLink may prevent interpretations from further misleading users. Instead, CCLink can remind users that the comment is of low quality to save their efforts.

Finally, we also find that CCLink increases the willingness of our participants in using NCG approaches. First, 18 of our participants, to some extent, express their negative attitudes toward NCG without interpretations. Their reasons include "poor accuracy" as we have discussed. They consider it more time-consuming in modifying the poor comments rather than writing them directly. Ten of them confirm their willingness to adopt NCG approaches in their production development practice, if being provided interpretations as CCLink. P7 suggests we integrate CCLink into popular IDEs so that developers can use it more conveniently. This confirms our argument that usability can be improved by not only increasing accuracy, but reasonable interpretations.

Code:

```
protected void unregisterListener ( ) {
    MetaDataProvider mdp = attributeType . getMetaDataProvider ( ) ;
    if ( mdp != null ) {
        mdp . removeMetaDataChangeListener ( this ) ;
    }
}
```

Auto-generated comment:

Unregister the metadata provider.

(a) Case 1

Code:

```
public void testcrlocationandtimevaluebutnolocation ( ) throws Exception {
    try {
        new TestHelper ( str_ , str_ , str_ , str_ ) . runTest ( ) ;
    } catch ( Exception e ) {
        assertTrue ( e . toString ( ) . contains ( GeospatialSearchException
. class . getName ( ) ) ) ;
        return ;
    }
    throw new RuntimeException ( str_ ) ;
}
```

Auto-generated comment:

Test the <UNK> and time value.

(b) Case 2

Code:

```
public Builder addShard ( IndexShardRoutingTable refData , ShardRouting shard ) {
    IndexShardRoutingTable indexShard = shards . get ( shard . id ( ) ) ;
    if ( indexShard == null ) {
        indexShard = new IndexShardRoutingTable . Builder ( refData . shardId ( ) )
. addShard ( new ShardRouting ( shard ) ) . build ( ) ;
    } else {
        indexShard = new IndexShardRoutingTable . Builder ( indexShard ) .
addShard ( new ShardRouting ( shard ) ) . build ( ) ;
    }
    shards . put ( indexShard . shardId ( ) . id ( ) , indexShard ) ;
    return this ;
}
```

Auto-generated comment:

Adds a new shard to the list at the specified index.

(c) Case 3

Fig. 11. Examples of interpretations produced by CCLink.

6 Further Discussions

6.1 Case Analysis

We now demonstrate some examples of links identified by CCLink. Fig. 11 shows three cases, each containing a Java function, the auto-generated comment, and the link identified by CCLink. We highlight each *focus* in the comments and the linked code with the same color.

The first case is a successful case where the interpretation helps correct the comment. The *focus*"unregister" is linked

to the function name "unregisterListener", and "metadata provider" is linked to the statement of calling the function "removeMetaDataChangeListener". With this interpretation, developers can realize that this function is used to unregister the metadata change listener, instead of the metadata provider. They can instantly correct the auto-generated comment.

The second case shows the effectiveness of CCLink even when the auto-generated comment is of poor quality. In this case, the comment contains "<UNK>", an unknown word. This is typically caused by the low frequency of the generated word or the anomalous behavior of the NMT model. Fortunately, CCLink can still link the "<UNK>" to the contributed code, which is "location" in the function name. With the help of CCLink, developers can quickly find what the unknown word means and correct the comment correspondingly. The other link of "time value" is also correct in this case, also helping them check the correctness of the *focus* quickly.

The code in the third case is more complicated than the previous two, and the link between code and comment is less obvious. This function is to add a shard at a specified index. In this case, the links identified by CCLink are not fully accurate. First, the *focus*"new shard" is linked to the statement of getting a new shard from the shard list. This is a correct link that helps developers understand the source of the"new shard". Another *focus*"specified index" is linked to the statement related to the shard ID, which is also a helpful link that facilitates a quick check. However, the *focus*"list" is linked to the function signature, which is not much related to the "list". The reason for this inaccurate link may due to the poor quality of the auto-generated comments, as we have discussed in Section 5.5.

The above example code-comment links can illustrate the use scenario of CCLink. We can see that with the help of CCLink, developers can examine the correctness of the auto-generated comments conveniently and make modifications more quickly. This can thus improve the usability of NCG approaches, and in turn the developers' efficiency in writing comments.

6.2 Implications to Future Work

In this work, we propose CCLink to interpret code comments generated by NCG approaches, as an attempt to improve the usability of NCG approaches. Next, we discuss the implications of our work to future work on automatic code comprehension. First, the key insight of our work is that the community should also focus on the interpretability of NCG approaches, as as to improve its usability. Existing NCG approaches are still far from applicable in practice due to their poor usability. In particular, it is hard to interpret the auto-generated comments for developers to correct the comments. Our work show that enhancing its interpretability is a promising research direction for NCG. We expect our interpretation framework CCLink can shed light to future research efforts in this direction.

Also, our empirical study indicates that the quality of auto-generated comments has an impact on the effectiveness of interpretation. So it is still important to improve the quality of auto-generated comments. Existing work mainly focuses on designing more sophisticated deep learning models. However, there may be other more effective ways to improve NCG approaches, e.g., building task-specific datasets and proper data preprocessing. This is still an open problem worthy of further exploration.

Another promising future work is to explore what kinds of interpretations are considered more helpful by the developers. In our empirical study, our investigation shows that most participants emphasize the importance of semantic similarity. In other words, they consider the interpretations with high semantic similarity to the *focuses* are more helpful. This is consistent with our metric for evaluating interpretation quality in Section 4.2. Nonetheless, comprehensive human-subject studies of this problem are of interest, calling for joint research efforts from both the software engineering community and the computer-supported cooperative work community. Example research questions include what phrases in the comments are more worthy of attention and which parts of code are more likely to generate valuable comments.

6.3 Threats to Validity

We also discuss the threats to the validity of our study and the measures we take to address them. First, datasets are critical considerations, which may mislead our findings. We consider two Java code corpora well-recognized in many existing studies. Both are based on real-world projects. Hence, they are good representatives of real code. We also carefully perform random sampling to avoid possible bias.

Second, it is difficult to model users' perception of interpretation quality with a quantitative indicator. To address this issue, we use Pearson correlation, the semantic correlation between a *focus* and its interpretation, to indicate interpretation quality. Our consideration is based on the well-accepted concept that code is also human-readable communication [11]. Meanwhile, such a method is also applied in existing work that evaluates the interpretation quality of DNN language models [50], [51].

Third, CCLink mainly uses `attendgru` in our study. A possible threat is whether CCLink can work with other approaches. In fact, CCLink is carefully designed without relying on the specifics of the NCG approaches. We also study CCLink by applying it to another NCG approach in our experimental study, which confirms our claim. In this work, we design and apply only three contribution mining methods (FreqCM, LimeCM, and AnchorCM). There may be many possible heuristics for result interpretation. Contribution mining can influence our results. Further possible enhancement can be applied. CCLink actually implements contribution mining as a plugin to facilitate further improvement.

Lastly, participants may bring bias to our results. For example, an inexperienced programmer may tend to rely more on auto-generated comments, and cannot precisely write comments. So we recruit 32 participants who have over five years of Java programming experience. We consider they are representatives of CCLink's target users. We also analyze the videos of experiments and confirm that they conduct the tasks smoothly as we expected. The results from the empirical study are reliable to produce our findings.

7 Related Work

7.1 Automatic Code Summarization

Automatic code summarization has long been studied to facilitate code comprehension. Many approaches have proposed to construct a set of complex rules, based on which comments can be generated with a template [4], [5]. Recent studies suggest that deep neural networks (DNNs) be applied to this task. Such neural comment generation (NCG) approaches automatically learn from existing code-comment data to produce comments for new code.

NCG for subroutines (e.g., function-level code snippets) is one widely-studied topic. Early studies treat source code as plain text. Iyer et al. [6] propose CODE-NN, which adopts LSTM networks with attention (a classic NMT model) to generate sentences that describe C# code snippets and SQL queries. Allamanis et al. [27] introduce an attentional convolutional neural network (CNN) to summarize source code snippets. Recent work also proposed to consider code structures. Hu et al. [7] propose DeepCom, using abstract syntax tree (AST) to annotate the words in Java methods as the input of NCG. CODE2SEQ [29] also leverages the syntactic structure of programming languages to better encode source code. LeClair et al. [8] propose a novel NMT model `ast-attendgru` that combines words from code with its AST to generate code- comments. Hu et al. [9] propose Hybrid-DeepCom, improving DeepCom with hybrid lexical and syntactical information.

Works related to NCG also include code change summarization. Jiang et al. [56] and Loyola et al. [57] apply NMT to automatically summarize code changes (i.e., commit messages). Liu et al. [58] further adopt an NMT model to automatically generate pull request (PR) descriptions based on the commit messages and the added source code comments in the PRs. Tufano et al. [59] conduct an investigation on the types of code changes that can be learned and applied automatically by NMT. Hoang et al. [60] propose CC2Vec, a DNN to produce distributed representations of code changes.

Existing learning-based approaches for code summarization have shown their advantages. However, they are not yet widely-used in practice largely due to their accuracy limitation and poor usability [8], [9]. This work aims to address the usability issue of NCG. Next, we will survey related work on software usability.

7.2 Software Usability

Software usability focuses on users' feelings about software, which is critical to software design. It has long been established that usability should be considered during iteration development [32] since it affects the usage of functionalities [35]. Several design principles for usability have been proposed [33], e.g., focusing on users as early as possible. In the 1990s, the importance of usability was further recognized and valued. For example, Mehlenbacher outlines the weakness and strengths of several usability evaluation approaches [61]. Usability became one of the key non-functional requirements of software [34]. In recent years, the study of usability gets more systematic. For example, Walenstein emphasizes software usability by boundary objects, which mitigates the gap between SE and HCI [62].

Actually, developers are users too [55], as developers usually adopt user mindsets when making user-related decisions during development [63] and using automatic tools [64]. Recent studies also focus on software usability of tools for developers. Piccioni et al. investigate the usability of APIs [65], as APIs are user interfaces of programming models for developers. Myers et al. summarizes several methods that can be used for user study and improving the usability of tools for developers [55]. Concerning automatic documentation, Wu et al. summarizes several challenges that hinder the usability of automatic documentation tools [66]. Our work also focuses on tool usability, which remains a critical research direction for the research community.

7.3 Interpretability of DNNs

To address the usability issue of NCG tools, we propose to enhance their interpretability, as existing studies show proper interpretations can improve the usability of such learning-based software [17], [18]. Although sophisticated DNNs have shown their effectiveness in many applications, it is notoriously hard to interpret how and why a DNN produces a specific result [20]. Despite many attempts in the literature, it remains an open problem [36], [37], [38], [39], [40], [41]. CCLink relies on model-independent result interpretation. Such interpretation infers why a result is produced, without examining the implementation details of the target model.

Existing work generally focuses on classification models. Poulin et al. [67] propose ExplainD to provide a graphical interpretation of classification results. Erik et al. [68] interpret individual classifications with game theory. These two approaches rely on specific types of data. LIME provides local model-agnostic interpretations of any classifier by constructing a linear model to locally fit a complex DNN model [17]. It does not depend on any specifics of data and models. A similar approach is Model Explanation System (MES) [69]. Ribeiro et al. [18] further propose Anchor, an extension of LIME based on decision rules. An anchor is a decision rule that leads to the result. LORE [70] is another local rule-based interpretation approach similar to Anchor. In this work, we apply LIME and Anchor as examples to interpret comments, which is the first exploration of applying DNN result interpretation to this task.

8 Conclusion

This work focuses on the usability of *Neural Comment Generation* (NCG). Instead of trying to improve the performance of NCG, we justify that existing NCG approaches suffer from poor usability in practice due to the lack of interpretability. In this paper, we design and implement CCLink to interpret the results of NCG approaches, which aim to find out the links between code and its auto-generated comment. Our study proves that CCLink can provide proper interpretations, which can help developers comprehend code and write better comments. We show that CCLink is a promising direction towards the practical application of NCG, which may shed light on this line of research.

References

[1] P. Hallam, "What do programmers really do anyway," *Microsoft Developer Netw. (MSDN) C# Compiler*, 2006.

[2] W. Maalej and H.-J. Happel, "Can development work describe itself?," in *Proc. IEEE 7th Work. Conf. Mining Softw. Repositories*, 2010, pp. 191–200.

[3] R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen, "BOA: A language and infrastructure for analyzing ultra-large-scale software repositories," in *Proc. ACM/IEEE 35th Int. Conf. Softw. Eng.*, 2013, pp. 422–431.

[4] E. Hill, L. Pollock, and K. Vijayshanker, "Automatically capturing source code context of NL-queries for software maintenance and reuse," in *Proc. ACM/IEEE 31st Int. Conf. Softw. Eng.*, 2009, pp. 232–242.

[5] S. Haiduc, J. Aponte, L. Moreno, and A. Marcus, "On the use of automated text summarization techniques for summarizing source code," in *Proc. IEEE 17th Work. Conf. Reverse Eng.*, 2010, pp. 35–44.

[6] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, "Summarizing source code using a neural attention model," in *Proc. 54th Annu. Meeting Assoc. Comput. Linguistics*, 2016, pp. 2073–2083.

[7] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation," in *Proc. IEEE 26th Conf. Prog. Comprehension*, 2018, pp. 200–210.

[8] A. LeClair, S. Jiang, and C. McMillan, "A neural model for generating natural language summaries of program subroutines," in *Proc. IEEE/ACM 41st Int. Conf. Softw. Eng.*, 2019, pp. 795–806.

[9] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation with hybrid lexical and syntactical information," *Empir. Softw. Eng.*, vol. 25, pp. 2179–2217, 2020.

[10] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," in *Proc. 3rd Int. Conf. Learn. Representations*, 2015, p. 15.

[11] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, "A survey of machine learning for big code and naturalness," *ACM Comput. Surv.*, vol. 51, no. 4, pp. 1–37, 2018.

[12] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "BLEU: A method for automatic evaluation of machine translation," in *Proc. 40th Annu. Meeting Assoc. Comput. Linguistics*, 2002, pp. 311–318.

[13] T. Kulesza, M. Burnett, W.-K. Wong, and S. Stumpf, "Principles of explanatory debugging to personalize interactive machine learning," in *Proc. 20th Int. Conf. Intell. User Interfaces*, 2015, pp. 126–137.

[14] E. Rader, K. Cotter, and J. Cho, "Explanations as mechanisms for supporting algorithmic transparency," in *Proc. CHI Conf. Hum. Factors Comput. Syst.*, 2018, pp. 1–13.

[15] J. Kunkel, T. Donkers, L. Michael, C.-M. Barbu, and J. Ziegler, "Let me explain: Impact of personal and impersonal explanations on trust in recommender systems," in *Proc. CHI Conf. Hum. Factors Comput. Syst.*, 2019, pp. 1–12.

[16] Q. Roy, F. Zhang, and D. Vogel, "Automation accuracy is good, but high controllability may be better," in *Proc. CHI Conf. Hum. Factors Comput. Syst.*, 2019, pp. 1–8.

[17] M. T. Ribeiro, S. Singh, and C. Guestrin, ""Why should I trust you?" Explaining the predictions of any classifier," in *Proc. 22nd ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining*, 2016, pp. 1135–1144.

[18] M. T. Ribeiro, S. Singh, and C. Guestrin, "Anchors: High-precision model-agnostic explanations," in *Proc. 32nd AAAI Conf. Artif. Intell.*, 2018, Art. no. 187.

[19] Y. Wu et al., "Google's neural machine translation system: Bridging the gap between human and machine translation," 2016, *arXiv:1609.08144*.

[20] R. Guidotti, A. Monreale, S. Ruggieri, F. Turini, F. Giannotti, and D. Pedreschi, "A survey of methods for explaining black box models," *ACM Comput. Surveys.*, vol. 51, no. 5, pp. 1–42, 2018.

[21] S. H. Tan, D. Marinov, L. Tan, and G. T. Leavens, "@tComment: Testing Javadoc comments to detect comment-code inconsistencies," in *Proc. IEEE 5th Int. Conf. Softw. Testing Verification Validation*, 2012, pp. 260–269.

[22] A. Louis, S. K. Dash, E. T. Barr, and C. Sutton, "Deep learning to detect redundant method comments," 2018, *arXiv: 1806.04616*.

[23] D. Chiang, "Hierarchical phrase-based translation," *Comput. Linguistics*, vol. 33, no. 2, pp. 201–228, 2007.

[24] M. Galley, M. Hopkins, K. Knight, and D. Marcu, "What's in a translation rule?," in *Proc. Hum. Lang. Technol. Conf. North Amer. Chapter Assoc. Comput. Linguistics*, 2004, pp. 273–280.

[25] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, 1997.

[26] B. Zhang, D. Xiong, and J. Su, "A GRU-gated attention model for neural machine translation," 2017, *arXiv: 1704.08430*.

[27] M. Allamanis, H. Peng, and C. Sutton, "A convolutional attention network for extreme summarization of source code," in *Proc. Int. Conf. Mach. Learn.*, 2016, pp. 2091–2100.

[28] P. Yin, B. Deng, E. Chen, B. Vasilescu, and G. Neubig, "Learning to mine aligned code and natural language pairs from stack overflow," in *Proc. IEEE/ACM 15th Int. Conf. Mining Softw. Repositories*, 2018, pp. 476–486.

[29] U. Alon, S. Brody, O. Levy, and E. Yahav, "code2seq: Generating sequences from structured representations of code," in *Proc. Int. Conf. Learn. Representations*, 2019, p. 22.

[30] S. Jiang, "Boosting neural commit message generation with code semantic analysis," in *Proc. IEEE/ACM 34th Int. Conf. Automated Softw. Eng.*, 2019, pp. 1280–1282.

[31] 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 *Proc. ACM/IEEE 33rd Int. Conf. Automated Softw. Eng.*, 2018, pp. 373–384.

[32] J. M. Carroll and M. B. Rosson, "Usability specifications as tool in iterative development," *Proc. Adv. Human-Comput. Interact.*, vol. 1, pp. 1–28, 1985.

[33] J. D. Gould and C. Lewis, "Designing for usability: Key principles and what designers think," *Commun. ACM*, vol. 28, no. 3, pp. 300–311, 1985.

[34] T. G. Kirner and A. M. Davis, "Nonfunctional requirements of real-time systems," *Adv. Comput.*, vol. 42, pp. 1–37, 1996.

[35] N. C. Goodwin, "Functionality and usability," *Commun. ACM*, vol. 30, no. 3, pp. 229–233, 1987.

[36] K. Xu et al., "Show, attend and tell: Neural image caption generation with visual attention," in *Proc. Int. Conf. Mach. Learn.*, 2015, pp. 2048–2057.

[37] B. Zhou, A. Khosla, A. Lapedriza, A. Oliva, and A. Torralba, "Learning deep features for discriminative localization," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2016, pp. 2921–2929.

[38] R. R. Selvaraju, M. Cogswell, A. Das, R. Vedantam, D. Parikh, and D. Batra, "Grad-CAM: Visual explanations from deep networks via gradient-based localization," in *Proc. IEEE Int. Conf. Comput. Vis.*, 2017, pp. 618–626.

[39] S. Bach, A. Binder, G. Montavon, F. Klauschen, K.-R. Müller, and W. Samek, "On pixel-wise explanations for non-linear classifier decisions by layer-wise relevance propagation," *PLoS One*, vol. 10, no. 7, 2015, Art. no. e0130140.

[40] G. Montavon, S. Lapuschkin, A. Binder, W. Samek, and K.-R. Müller, "Explaining nonlinear classification decisions with deep taylor decomposition," *Pattern Recognit.*, vol. 65, pp. 211–222, 2017.

[41] A. Shrikumar, P. Greenside, and A. Kundaje, "Learning important features through propagating activation differences," in *Proc. 34th Int. Conf. Mach. Learn.*, 2017, pp. 3145–3153.

[42] S. Siddiqi and A. Sharan, "Keyword and keyphrase extraction techniques: A literature review," *Int. J. Comput. Appl.*, vol. 109, no. 2, pp. 18–23, 2015.

[43] T. Cerny, M. J. Donahoo, and M. Trnka, "Contextual understanding of microservice architecture: Current and future directions," *ACM SIGAPP Appl. Comput. Rev.*, vol. 17, no. 4, pp. 29–45, 2018.

[44] A. Endert, S. Fox, D. Maiti, S. Leman, and C. North, "The semantics of clustering: Analysis of user-generated spatializations of text documents," in *Proc. Int. Work. Conf. Adv. Vis. Interfaces*, 2012, pp. 555–562.

[45] S. Rose, D. Engel, N. Cramer, and W. Cowley, "Automatic keyword extraction from individual documents," *Text Mining: Appl. Theory*, vol. 1, pp. 1–20, 2010.

[46] J. Han, H. Cheng, D. Xin, and X. Yan, "Frequent pattern mining: Current status and future directions," *Data Mining Knowl. Discov.*, vol. 15, no. 1, pp. 55–86, 2007.

[47] R. Agrawal and R. Srikant, "Fast algorithms for mining association rules," in *Proc. 20th Int. Conf. Very Large Data Bases*, 1994, pp. 487–499.

[48] E. Kaufmann and S. Kalyanakrishnan, "Information complexity in bandit subset selection," in *Proc. 26th Annu. Conf. Learn. Theory*, 2013, pp. 228–251.

[49] J. A. Jones, M. J. Harrold, and J. Stasko, "Visualization of test information to assist fault localization," in *Proc. IEEE 24th Int. Conf. Softw. Eng.*, 2002, pp. 467–477.

[50] W. J. Murdoch, P. J. Liu, and B. Yu, "Beyond word importance: Contextual decomposition to extract interactions from LSTMs," in *Proc. Int. Conf. Learn. Representations*, 2018, p. 15.

[51] X. Jin, J. Du, Z. Wei, X. Xue, and X. Ren, "Towards hierarchical importance attribution: Explaining compositional semantics for neural sequence models," in *Proc. Int. Conf. Learn. Representations*, 2020, p. 15.

[52] J. Benesty, J. Chen, Y. Huang, and I. Cohen, "Pearson correlation coefficient," in *Noise Reduction Speech Process.*. Berlin, Germany: Springer, 2009, pp. 1–4.
[53] J. Pennington, R. Socher, and C. D. Manning, "GloVe: Global vectors for word representation," in *Proc. Conf. Empir. Methods Natural Lang. Process.*, 2014, pp. 1532–1543.
[54] P. Biernacki and D. Waldorf, "Snowball sampling: Problems and techniques of chain referral sampling," *Sociol. Methods Res.*, vol. 10, no. 2, pp. 141–163, 1981.
[55] B. A. Myers, A. J. Ko, T. D. LaToza, and Y. Yoon, "Programmers are users too: Human-centered methods for improving programming tools," *Computer*, vol. 49, no. 7, pp. 44–52, 2016.
[56] S. Jiang, A. Armaly, and C. McMillan, "Automatically generating commit messages from diffs using neural machine translation," in *Proc. IEEE/ACM 32nd Int. Conf. Automated Softw. Eng.*, 2017, pp. 135–146.
[57] P. Loyola, E. Marrese-Taylor, and Y. Matsuo, "A neural architecture for generating natural language descriptions from source code changes," in *Proc. 55th Annu. Meeting Assoc. Comput. Linguistics*, 2017, pp. 287–292.
[58] Z. Liu, X. Xia, C. Treude, D. Lo, and S. Li, "Automatic generation of pull request descriptions," in *Proc. IEEE/ACM 34th Int. Conf. Automated Softw. Eng.*, 2019, pp. 176–188.
[59] M. Tufano, J. Pantiuchina, C. Watson, G. Bavota, and D. Poshyvanyk, "On learning meaningful code changes via neural machine translation," in *Proc. IEEE/ACM 41st Int. Conf. Softw. Eng.*, 2019, pp. 25–36.
[60] T. Hoang, H. J. Kang, J. Lawall, and D. Lo, "CC2Vec: Distributed representations of code changes," in *Proc. ACM/IEEE 42nd Int. Conf. Softw. Eng.*, 2020, pp. 518–529.
[61] B. Mehlenbacher, "Software usability: Choosing appropriate methods for evaluating online systems and documentation," in *Proc. 11th Annu. Int. Conf. Syst. Documentation*, 1993, pp. 209–222.
[62] A. Walenstein, "Finding boundary objects in SE and HCI: An approach through engineering-oriented design theories," in *Proc. ICSE Workshop SE-HCI*, 2003, pp. 92–99.
[63] T. Roehm, R. Tiarks, R. Koschke, and W. Maalej, "How do professional developers comprehend software?," in *Proc. IEEE 34th Int. Conf. Softw. Eng.*, 2012, pp. 255–265.
[64] B. Lin, A. Zagalsky, M.-A. Storey, and A. Serebrenik, "Why developers are slacking off: Understanding how software teams use slack," in *Proc. 19th ACM Conf. Comput. Supported Cooperative Work Social Comput. Companion*, 2016, pp. 333–336.
[65] K. Arnold, "Programmers are people, too: Programming language and API designers can learn a lot from the field of human-factors design," *Queue*, vol. 3, no. 5, pp. 54–59, 2005.
[66] S. Wu, Y. Zhou, and X. Wang, "Exploring user experience of automatic documentation tools," in *Proc. Extended Abstr. CHI Conf. Hum. Factors Comput. Syst.*, 2021, pp. 1–6.
[67] B. Poulin et al., "Visual explanation of evidence with additive classifiers," in *Proc. Nat. Conf. Artif. Intell.*, 2006, Art. no. 1822.
[68] I. Kononenko et al., "An efficient explanation of individual classifications using game theory," *J. Mach. Learn. Res.*, vol. 11, no. Jan., pp. 1–18, 2010.
[69] R. Turner, "A model explanation system," in *Proc. IEEE 26th Int. Workshop Mach. Learn. Signal Process.*, 2016, pp. 1–6.
[70] R. Guidotti, A. Monreale, S. Ruggieri, D. Pedreschi, F. Turini, and F. Giannotti, "Local rule-based explanations of black box decision systems," 2018, *arXiv: 1805.10820*.

Shuyao Jiang received the BSc and MEng degrees from Fudan University, in 2019 and 2022, respectively. She is currently working toward the PhD degree with the Department of Computer Science and Engineering, Chinese University of Hong Kong. Her research interests include software engineering and runtime systems.

Jiacheng Shen received the BSc degree from Fudan University, in 2020. He is currently working toward the PhD degree with the Department of Computer Science and Engineering, Chinese University of Hong Kong. His research interests include distributed computing and cloud computing.

Shengnan Wu received the BSc and MEng degrees from Shanghai International Studies University, in 2016 and 2019, respectively. She is currently working toward the PhD degree with the School of Computer Science, Fudan University. Her research interests include software engineering and computer-supported cooperative work.

Yu Cai received the BSc and MEng degrees from Fudan University, in 2019 and 2022, respectively. He is currently working as an engineer with Tencent, China. His research interests include mobile computing and UI testing.

Yue Yu is an associate professor with the College of Computer, National University of Defense Technology (NUDT), and technical committee member of OpenI community. His research findings have been published on *IEEE Transactions on Software Engineering*, *ACM Transactions on Software Engineering and Methodology*, CHI, CSCW, ICSE, FSE, ASE etc. His current research interests include software engineering, data mining, and computer-supported cooperative work.

Yangfan Zhou received the BSc degree from Peking University, in 2000, and the MPhil and PhD degrees from the Chinese University of Hong Kong, in 2006 and 2009, respectively. He is currently a professor with Fudan University. Before joining Fudan, he was a research staff member with the Chinese University of Hong Kong from 2009 to 2014. His research interests include software engineering and computer-supported cooperative work.