Huang Y, Jia N, Zhou HJ *et al.* Learning human-written commit messages to document code changes. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 35(6): 1258–1277 Nov. 2020. DOI 10.1007/s11390-020-0496-0

Learning Human-Written Commit Messages to Document Code Changes

Yuan Huang[1], Nan Jia[2], Hao-Jie Zhou[1], Xiang-Ping Chen[3,*] *Member, IEEE*
Zi-Bin Zheng[1], *Senior Member, IEEE*, and Ming-Dong Tang[4,5], *Member, ACM, IEEE*

[1] *National Engineering Research Center of Digital Life, School of Data and Computer Science, Sun Yat-sen University Guangzhou 510006, China*

[2] *School of Information Engineering, Hebei GEO University, Shijiazhuang 050031, China*

[3] *Guangdong Key Laboratory for Big Data Analysis and Simulation of Public Opinion, School of Communication and Design, Sun Yat-sen University, Guangzhou 510006, China*

[4] *School of Information Science and Technology, Guangdong University of Foreign Studies, Guangzhou 510006, China*

[5] *Guangdong Key Laboratory of Big Data Analysis and Processing, Guangzhou 510006, China*

E-mail: huangyjn@gmail.com; jianan_0101@163.com; zhouhj8@mail2.sysu.edu.cn
{chenxp8, zhzibin}@mail.sysu.edu.cn; mdtang@gdufs.edu.cn

Received April 5, 2020; revised October 15, 2020.

Abstract Commit messages are important complementary information used in understanding code changes. To address message scarcity, some work is proposed for automatically generating commit messages. However, most of these approaches focus on generating summary of the changed software entities at the superficial level, without considering the intent behind the code changes (e.g., the existing approaches cannot generate such message: "fixing null pointer exception"). Considering developers often describe the intent behind the code change when writing the messages, we propose ChangeDoc, an approach to reuse existing messages in version control systems for automatical commit message generation. Our approach includes syntax, semantic, pre-syntax, and pre-semantic similarities. For a given commit without messages, it is able to discover its most similar past commit from a large commit repository, and recommend its message as the message of the given commit. Our repository contains half a million commits that were collected from SourceForge. We evaluate our approach on the commits from 10 projects. The results show that 21.5% of the recommended messages by ChangeDoc can be directly used without modification, and 62.8% require minor modifications. In order to evaluate the quality of the commit messages recommended by ChangeDoc, we performed two empirical studies involving a total of 40 participants (10 professional developers and 30 students). The results indicate that the recommended messages are very good approximations of the ones written by developers and often include important intent information that is not included in the messages generated by other tools.

Keywords commit message recommendation, code syntax similarity, code semantic similarity, code change comprehension

1 Introduction

In software maintenance, understanding code changes costs developers most of their time [1, 2]. These code changes are often organized and saved as commits in version control systems (e.g., Git). Normally, a message in natural language is written by the developer for each commit to help understanding the changes [3, 4].

Regular Paper

Special Section on Software Systems 2020

A preliminary version of the paper was published in the Proceedings of ESEM 2017.

This work was (partially) supported by the Key-Area Research and Development Program of Guangdong Province of China under Grant No. 2020B010164002, the National Natural Science Foundation of China under Grant Nos. 61902441, 61722214 and 61976061, the China Postdoctoral Science Foundation under Grant No. 2018M640855, the Fundamental Research Funds for the Central Universities of China under Grant Nos. 20wkpy06 and 20lgpy129, and the Opening Project of Guangdong Key Laboratory of Big Data Analysis and Processing under Grant No. 202003.

*Corresponding Author

The previous research shows that commit messages expound the rationale behind a code change, and they are used as the most important way to understand the code change [5, 6]. However, many commits in different projects still lack messages. Maalej and Happel [7] analyzed more than 600 000 commit messages demonstrating that 10% of the messages are empty, and another study of more than 23 000 projects by Dyer *et al.* [8] showed that 14% of the commit messages are empty.

Therefore, much work has been proposed for automatically generating commit messages [9–15]. Linares-Vàsquez *et al.* [9] presented ChangeScribe to automatically complement the commit messages. ChangeScribe firstly identifies the commit stereotype, as well as the impact set of code changes, and then generates commit messages according to the predefined templates. Moreno *et al.* [10, 11] introduced ARENA for the automatic release comment generation. ARENA identifies different types of changes between two releases, and generates a comment for the code changes via code summarization methods [16, 17]. These approaches use predefined templates to define the skeleton of the commit messages and instantiate these templates via program analysis. However, these approaches cannot summarize the intent of the code change. For example, Fig.1 shows the generated messages for commit #32478 by using ChangeScribe [9], a state-of-the-art approach, which employs a hierarchical style to describe the code changes occurring in the software entities, i.e., from packages to classes, and classes to methods, etc. Although the generated message covers most of the code changes, it misses the most important piece of information — the intent behind the code changes [5, 13]. On the other hand, the commit message written by the developer (Fig.1, Developer) shows exactly the intent of the change, "fixing null point exception", and nothing else.

We observe that version control systems such as Git save a huge number of commit messages written by developers which usually describe the change intents. Therefore, we try to "reuse" the existing messages in version control systems to automatically recommend the message for a target commit in this paper. Our approach, named ChangeDoc, is based on the code similarity between commits for message recommendation. More specifically, the commit in Git mainly consists of the changed code fragment as well as the corresponding message written by the developer [18, 19]. For a target commit, if its changed code fragment is similar to that of a commit in the version control system and these two commits may involve the similar code change, then the message of the commit in the version control system may be suitable for the target commit.

It is not easy to utilize existing commit messages to automate the message recommendation. First, we need to locate a commit from the version control system with similar changed code fragments to the target commit. To do this, a keyword-based search approach is a possible way. However, in some cases, two similar code fragments are rendered differently, which makes the keyword-based search approach useless. The changed code fragments in commits 1 and 2 (i.e., in Fig.2) implement the functionality of array element sorting. But it is difficult to retrieve commit 2 from commit 1 by using a keyword-based search approach, as these two fragments involve few intersecting keywords.

To overcome this problem, we introduce code syntactic information to measure the similarity of code fragments in addition to considering the intersecting keywords in the source code (i.e., semantic information). The semantic information is extracted from the

```
    String uri = nc.getUri();
    if(INHERIT.equals(uri)){
-:  String def = nc.getSource();
-:  if(def != null){
-:    uri = def;
+:  if(nc.getSource() != null){
+:    String def = nc.getSource();
+:    if(def != null){
+:      uri = def;
      }
    }
```

(a)

ChangeScribe:
This is a small modifier commit, which is mainly:
1. change package *plugins.XML.trunk.xml.parser*
1.1 modified to *SchemaTCompletion.java*
1.1.1 add statement at *getNamespace*()

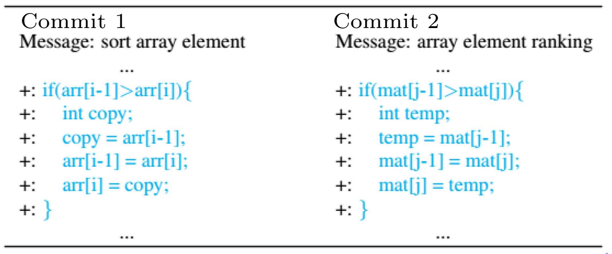
(b)

Developer:
Fix null pointer exeption (NPE) in *getNamespace*()

(c)

Fig.1. Commit messages generated by ChangeScribe and the developer. (a) Changed code fragment in commit #32478. (b) Messages generated by ChangeScribe. (c) Messages generated by the developer.

identifiers in source code, while the code syntactic information is obtained by analyzing the abstract syntax tree of code fragments. Our goal is to obtain a syntactic sequence for each code fragment, and this sequence represents the syntax types of the code lines sequentially composing of the fragment. If two syntactic sequences have a high degree of similarity, we can infer that the corresponding code fragments have a higher syntactic similarity. As a result, the retrieve task illustrated in Fig.2 will be easily solved if we consider the code syntactic similarity. However, it is not feasible to directly construct such a syntactic sequence for a code fragment, because a code line may contain several different types of syntax. To overcome this problem, we further fuse the syntax types after a hash operation.



```
Commit 1                              Commit 2
Message: sort array element           Message: array element ranking
        ...                                   ...
+: if(arr[i-1]>arr[i]){               +: if(mat[j-1]>mat[j]){
+:    int copy;                       +:    int temp;
+:    copy = arr[i-1];                +:    temp = mat[j-1];
+:    arr[i-1] = arr[i];              +:    mat[j-1] = mat[j];
+:    arr[i] = copy;                  +:    mat[j] = temp;
+: }                                  +: }
        ...                                   ...
```

Fig.2. Commits with similar changed code fragments.

Besides, to further improve the accuracy of ChangeDoc for commit message recommendation, we utilize the fact that a commit usually contains the source code of current and previous versions. The current version of commit code represents the code state after update (e.g., the code lines in cyan in Fig.1), while the previous version of commit code represents the code state before update (e.g., the code lines in magenta in Fig.1). If the codes of two commits have higher similarities in both their current and previous versions, they are likely to be modified toward the same goal (because similar changes were made to the similar code fragments in two commits). As a result, the message describing the code change in one commit is suitable for describing the code change in another commit. Four types of similarities, i.e., syntax, semantic, pre-syntax, and pre-semantic similarities, are introduced to measure the similarity of two commits before and after update. Syntax and semantic similarities are used to evaluate the syntactic and semantic similarities of code fragments of two commits after change, while pre-syntax and pre-semantics are used to measure the similarities of the code fragments of two commits before update.

We evaluate ChangeDoc in an experiment involving 10 projects with 1 000 commits. ChangeDoc is able to recommend messages for 52.4% of the commits, and 21.5% of the recommended messages can be directly used without modification, and 62.8% require minor modifications. In a summary, our contributions are as follows. 1) To our knowledge, ChangeDoc is the first one that mines version control system to recommend messages for commits. 2) The recommended messages by ChangeDoc can describe the intent behind the code changes, which the traditional methods cannot. 3) We evaluate our approach with real-world projects from SourceForge, and manually verify the correctness of the recommended commit messages. Our evaluation shows that ChangeDoc is effective in recommending commit messages, and can complement existing message generation approaches. 4) Two empirical studies involving a total of 40 participants are conducted to evaluate the quality of the recommended messages. The obtained results demonstrate that the messages recommended by ChangeDoc outperform those created by ChangeScribe in terms of conciseness, expressiveness, and preciseness. To facilitate research and application, our ChangeDoc[①] is available.

Compared with our previous work[20], the paper has the following new contributions. Firstly, we optimize the similarity calculation algorithm. For a commit containing multiple changed code fragments across multiple classes, our new algorithm identifies a salient class (which reflects the main intent of the change in a commit) in the commit, and uses the changed code fragments in the salient class instead of all the ones in the commit to calculate the similarity of commits. This optimization achieves a better result of recommending reusable messages. Secondly, we add and analyze datasets from another three software systems (i.e., UNICORE, Makagiga, and Kablink) for the message recommendation, and use a total of 10 subject projects in our experiment. Thirdly, we investigate an additional research question (RQ_4) in our evaluation that compares ChangeDoc with ChangeScribe[9], and find that the messages recommended by ChangeDoc can provide more important information than those created by ChangeScribe.

In the following parts of the paper, we start by introducing related work, summarizing the techniques for code change message generation, and providing an overview of the known methods for source code summarization. Then, in Section 3, we describe the proposed

[①]https://github.com/zhouhj8/DSCS.git, Sept. 2020.

method. We describe the setups and results of case study in Section 4. We outline a discussion regarding the θ choice effect in Section 5. We discuss the threats to validity in Section 6, and conclude the paper in Section 7.

2 Related Work

Most existing code change messages generating approaches are based on source code summarization techniques, which will be reviewed in this section. Therefore, we further survey the techniques for source code summarization. In addition, we discuss the techniques regarding the code clone, which is one of the core techniques for building ChangeDoc.

Code Change Message Generation. Recently, a number of researchers proposed to automatically generate code change messages. According to the granularities of description content, we have divided the message generation approaches into three levels: commit level, release level , and single code change.

The work proposed by Linares-Vàsquez *et al.*[9, 21], Shen *et al.*[12] and Buse and Weimer[13] generated the commit-level code change message. Linares-Vàsquez *et al.*[9, 21] presented an approach, ChangeScribe, to generate commit messages based on commit stereotype. ChangeScribe first extracts the stereotype, the type and the impact set of a commit by analyzing corresponding source code changes and the abstract syntax trees. Then it fills predefined templates with the extracted information to document this commit. Shen *et al.*[12] proposed an approach to automatically generate the "what" and "why" information for software changes, which is similar to ChangeScribe[9], but it constrains the length of the generated message by removing the repeated information in the change. Both the two methods analyze the commit's stereotype to generate commit messages. Buse and Weimer[13] proposed DeltaDoc to generate the summary of code commit. DeltaDoc obtains path predicates by symbolically executing source code changes and then generates commit messages using a set of predefined rules and transformations.

Moreno *et al.*[10, 11] proposed ARENA to generate code change messages at the release level. ARENA combines multiple kinds of changes, e.g., changes to source code, libraries, documentation and licenses, with issues from software repositories to generate release notes. In addition, there are some researchers focusing on generating messages for single code change[14, 15]. For example, to answer why a change happened, Rastkar and Murphy[14] proposed an approach to extract the motivational information of commits from multiple relevant documents. Parnin and Görg[15] proposed to generate code change description via decompiling bytecode instructions, and the generated summary includes the information that is harder to obtain with textual representation. Overall, while the existing approaches generate different types of code summaries, they cannot describe the intent behind a code change.

Most of the mentioned approaches are based on the predefined-templates[9–11, 13]. Recently, Jiang and McMillan[22] found that most of the commit messages often begin with a verb followed by a direct object in their empirical study. Then, as the first step[22], they used a verb to label each commit diff via employing a machine learning algorithm. As the second step[23], they adapted a neural machine translation algorithm (i.e., NMT) to automatically translate diffs into commit messages with a format of "verb+object". They trained an NMT algorithm using pairs of diffs and commit messages from 1 000 popular projects on GitHub. Essentially, the work of Jiang and Armaly[23] is still template-based, in which they used a "verb+object" template and employed machine learning to fill the template. However, due to the limitation of the dataset, only a fixed format message can be generated for each type of commits by their method.

Hoang *et al.*[24] proposed a deep learning architecture, namely CC2Vec, that learns distributed representations of code changes guided by the semantic meaning contained in log messages, and then CC2Vec is applied to the task of log message generation, and the result indicates that CC2Vec outputs the state-of-the-art approach. Xu *et al.*[25] proposed CoDiSum to address the commit message generation problem, and they jointly modelled the code structure and code semantics of the code changes to learn code representations. Liu *et al.*[26] proposed NNGen to generate a commit message for a code change. NNGen first extracts code changes from the training set, and then calculates the cosine similarity between the vector of the new code change and the vector of each code change in the training set. A commit message of the code change with the highest BLEU-4 score to the new code change is reused as the commit message of the new code change. Nie *et al.*[27] proposed a contextualized code representation learning method for commit message generation (i.e., CoreGen). CoreGen learns contextualized code representation which exploits the contextual information behind code commit sequences. Liu *et al.*[28] proposed

ATOM to encode AST paths of diffs for code representation to generate commit messages. ATOM provides a hybrid ranking module to enhance the output of generation modules, by providing the most accurate commit messages among the generated and retrieved results. The results demonstrate that ATOM increases the performance of the state-of-the-art models by 30.72% in terms of BLEU-4.

Source Code Summarization. Most of the mentioned methods for commit message generation [9–12, 14] are based on source code summarization techniques.

One of the code summarization techniques adapts a template-based framework to summarize source code [26]. Specifically, they first select important content from source code and then transform the selected content into natural language descriptions through predefined templates. For example, to summarize Java methods, the framework proposed by Sridhara *et al.* [16] first identifies significant statements of a Java method according to structural and linguistic clues and then expresses extracted content in natural language using predefined text templates. McBurney and McMillan [29] analyzed the method invocations to summarize the context of why the method exists or what role it plays in the software. Moreno *et al.* [17] generated human readable summaries regarding the responsibilities of the classes.

The second method for code summarization is based on the information retrieval techniques. These methods generate code comments by searching the similar code snippets from the code base. For example, Wong *et al.* [30] proposed a novel method to automatically generate code comments by mining large-scale Q&A data from StackOverflow. Meanwhile, Wong *et al.* [31] applied code clone detection techniques to discover similar code segments in software repositories and used existing comments to describe similar code segments. Haiduc *et al.* [32, 33] generated term-based summaries for classes and methods by employing information retrieval techniques, and they also compared the suitability of several summarization techniques.

In addition, some work leverages deep learning techniques to summarize source code. These approaches use deep learning to train probabilistic models for the code summary generation. Iyer *et al.* [34] used the LSTM model to design a code summary automatic generation method for C# code and SQL queries. Allamanis *et al.* [35] used convolutional neural network to summarize the Java code into short, name-like summaries (average in three words). Hu *et al.* [36] took structural and semantic information from the abstract syntax tree, converted it into sequence information, and then used the machine translation model to translate the code into summary. Since then, Hu *et al.* [37] took code API information into consideration, which further improves the accuracy of code summary generation. In our work, we directly utilize the existing commit messages instead of the source code summarization techniques to generate commit messages, different from all the mentioned methods.

Code Clone Detection. In order to recommend a suitable commit message to the target commit, we need to retrieve a commit with similar changed code segments to the target commit from the local repository. Actually, discovering the similar changed code segments in commits is a code clone detection problem. There are many ways to detect code clone. One of the representative methods is comparing the abstract syntax trees to calculate how similar two code fragments are. This method is a tree-based method [38]. It firstly calculates the similarities of the subtrees until the similarity of two entire trees can be acquired. The tree-based method may achieve an exponential time complexity [39], which is not suitable in our search-based scenario.

Another representative method to detect code clone is a token-based method [40]. This method firstly generates tokens for each code line. A code fragment is represented as a token sequence. The syntax similarity of two code fragments can be calculated by finding the longest matched token sequence of two fragments. The token-based approach is inherently low-cost [41, 42]. It works faster because it only needs to transform the source code into tokens, without the need to construct ASTs. As a result, the time complexity of the token-based method is $O(n \times m)$ [40]. Then, we choose it to calculate the code syntax similarity in this paper.

3 Approach

3.1 Approach Overview

ChangeDoc firstly collects the commits from the projects' version control systems, and then stores these commits in a local repository. After that, for a target commit that needs to be a recommended comment, our approach retrieves the local repository taking the source code of the target commit as input and yields and ranks the similar commits in a list. Finally, the message of the most similar commit is recommended to the target commit. The overview of the proposed approach is shown in Fig.3.

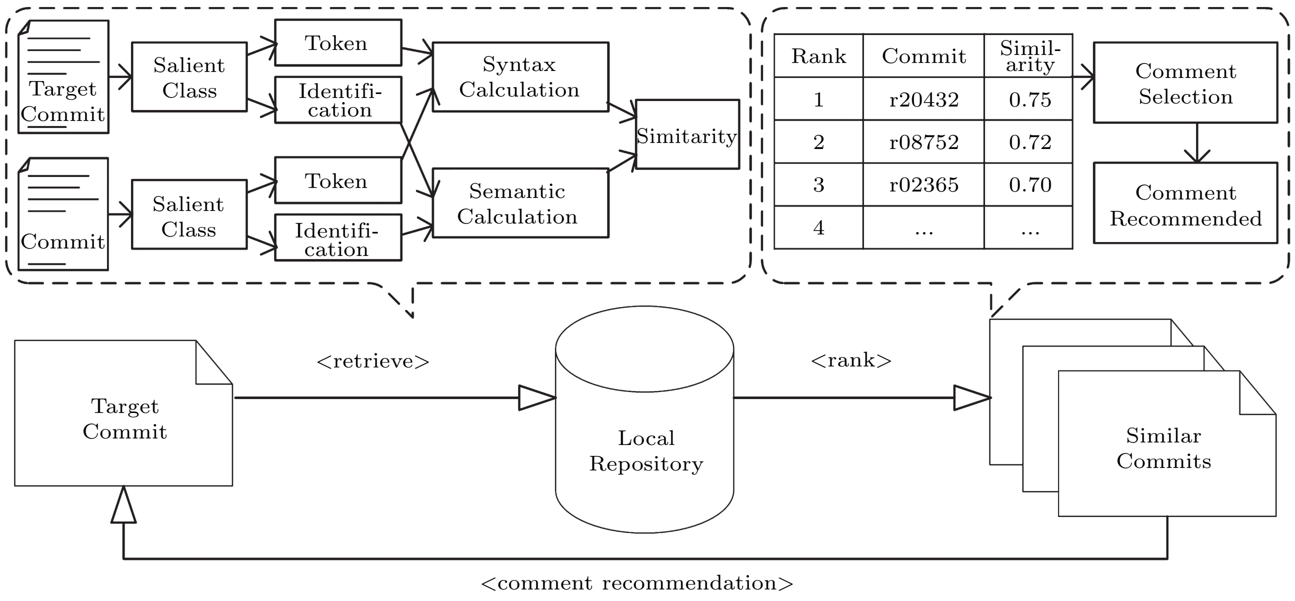


Fig.3. Overview of the proposed approach.

Specifically, for the target commit and the commits in the local repository, our approach firstly identifies their core modified classes (referred to as salient class). For each commit, our approach employs the tool ChangeDistiller [43] to detect the changed code fragment in the salient class. It utilizes the AST tree of the source code of the changed code fragments to analyze the syntax structures, and generates syntactic tokens for the changed code fragment. In addition, the significative identifiers (denoted by `identfi` in Fig.3) in changed code fragment are extracted for analyzing the semantic information of the source code. For two commits, their syntax and semantic similarities are calculated based on the generated tokens and extracted identifiers. At last, a list of commits is generated according to the syntax and semantic similarities, and the message of the commit at the top of the list (i.e., most similar to the target commit) is recommended as the message of the target commit.

3.2 Similar Commits Retrieving

The goal of ChangeDoc is to retrieve similar commits from the existing commit repository and further to recommend the messages of the similar commits to the target commit. In this paper, ChangeDoc utilizes the code syntax and semantics to calculate the similarities between commits, due to the rich syntactic and semantic information contained in the source code [44]. To find a suitable message for the target commit, it actually finds a commit in the repository with a similar code change to the target commit, and then the message of the commit in the repository might be reused by the code change of the target commit. Therefore, ChangeDoc firstly extracts the changed code fragments from commits, and then applies syntax and semantic analysis algorithms to further measure the similarity between the changed code fragments of two commits.

3.2.1 *Commit Pre-Processing*

Salient Changed Class Identification. A commit often involves multiple classes with multiple changed code fragments [45]. Different changes may have different levels of significance: some may be the salient changes reflecting the intent of the developers, while other changes may be dependent changes that are required by the salient change. In our previous study [46], we found that a single commit often includes a class that is saliently modified, which usually represents the change intent of the commit, and the rest of the classes in the commit are dependency modifications. If we take all the changed code fragments (in salient changed class and dependency modification classes) into the similarity calculation, the changed code fragment representing the change intent will be "diluted".

Therefore, we propose to consider only the changes in the salient class for similarity calculation, i.e., we will only compare the changes in salient classes when we calculate the similarity of two commits. The salient changed class identification algorithm has been proposed in our previous study [46], known as ISC, and we directly employ ISC to identify the salient changed class of each commit in this paper. ISC tries to use the structural coupling information between classes, update the degree of a class, and commit type information to distinguish the salient and non-salient changed classes. ISC achieves an accuracy of 87%. The detailed introduction of ISC can be found in [46]. In the case study section, we will further evaluate the performances of ChangeDoc when applying original (without salient changed class identification, referred to as original similarity calculation algorithm) and improved (with salient changed class identification, referred to as improved similarity calculation algorithm) similarity calculation algorithms.

Changed Code Identification. The algorithm for identifying changed code fragment is based on ChangeDistiller [43]. ChangeDistiller compares the abstract syntax trees of two versions of the code to find out the difference, which has been demonstrated effective on many problems [47–49]. Specifically, ChangeDistiller uses the statement as the smallest AST node and categorizes source code changes into four types of elementary tree edit operations, namely, inserting, deleting, moving and updating. Then, the refactoring-based operations such as condition expression change, method renaming, parameter deletion, ordering change, type change, statement insertion, parent changes, etc., can be detected.

For a commit, we firstly identify its salient changed class, and then use ChangeDistiller to identify the changed code fragments in the salient class. Because ChangeDistiller can identify the start code line and the end code line of a changed code fragment from the source code of the salient class, we can extract the changed code fragment in the salient class according to its start and end code lines. It is worth noting that a salient class may involve multiple changed code fragments, which are scattered in the salient class, and not connected [46]. To make it easier to compare the similarity of two salient classes later, all the changed code fragments in a salient class are spliced together to form a larger changed code fragment. As a result, each commit only corresponds to a changed code fragment.

3.2.2 Syntax Similarity Analysis

Code Tokenizing. In this paper, we extract code syntactic types to tokenize the code lines, referred to as syntactic tokens. We mainly focus on 96 types of code syntactic tokens (e.g., IfStatement, MethodInvocation, SwitchStatement, ThrowStatement, CastExpression, VariableDeclaration, ClassInstanceCreation, etc.) [50, 51]. For each type of syntactic tokens, we first get its start line and end line in the source code via parsing the abstract syntax tree, and then count the syntactic tokens involved by each code line. For example, Fig.4 shows the process of code tokenizing, and $<18, 20>$ in step 2 represents the start line and the end line of token IfStatm (i.e., IfStatement) in the source code, and step 3 shows the syntactic tokens involved by each code line. It is worth noting that a code line may contain several syntactic tokens and the syntactic tokens may be of different types [52], e.g., the code line "`if(epu.getName() == null)`" in Fig.4 contains four syntactic tokens: IfStatement, ConditionalExpression, MethodInvocation, and NullLiteral.

Hash Sequence. A single code line may contain more than one syntactic token, which makes it difficult to quickly calculate the similarity of two code fragments via comparing the syntactic token. One possible solution for this issue is to digitize the syntactic tokens, and then the syntactic tokens contained in a code line can be fused into a uniform number. Specifically, we firstly get the syntactic tokens of each code line in the changed code fragments. Then, each type of syntactic tokens is defined to correspond to a unique hash value (10 digits). If a code line contains only one type of tokens, its hash value is the same as the hash value of this kind of tokens. If a code line involves multiple types of syntactic tokens, the hash values of each type of syntactic tokens are added together to generate a new hash. At last, a code line is mapped to a hash value, and a changed code

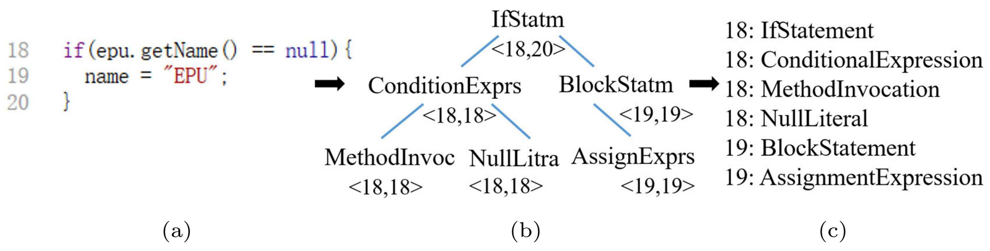


Fig.4. Code tokenizing. (a) Step 1. (b) Step 2. (c) Step 3.

fragment is mapped to a sequence of hash values. We calculate the syntactic similarity of two changed code fragments based on the hash sequences.

Matching Algorithm. We employ the token-based code clone detection method[40] to discover the similar changed code in this paper. For the corresponding hash sequences of two changed code fragments, we try to find their longest matching subsequence to measure the syntactic similarity of these two code fragments. We employ the algorithm proposed by Wettel and Marinescu[40] to find out the longest matching subsequence. The difference is that we utilize the abstract syntax tree to parse and tokenize the code to generate the hash sequence, while Wettel and Marinescu[40] used regular expressions in identifying the elements in the source code.

As shown in Algorithm 1, the matching algorithm includes four steps. Firstly, the hash values in hash sequence 1 (corresponding to code fragment 1) are compared with the hash values in hash sequence 2 (corresponding to code fragment 2). A matrix ***Mtrx*** is used to store the result. A matrix cell $Mtrx[i, j]$ stores the comparison result of the i-th hash value in sequence 1 and the j-th hash value in sequence 2. $Mtrx[i, j] = 1$ means the two hashes are matched. Secondly, we traverse the matrix from the upper left and examines each matched cell. From a matched cell, we further extend up until the first unmatched cell in the main diagonal direction. These continuously matched cells form a subsequence. At last, we find out all of the matched subsequences in the matrix. Thirdly, we identify the gap between any two subsequences in the matrix. The gap is the unmatched cells in the matrix. If the gap is less than or equal to 2[40], these two subsequences are merged to form a longer one, which is added into the subsequence list. The gaps are repeatedly checked until we traverse all the subsequences and then we can find a longest subsequence at last. Fourthly, the length of the longest subsequences is divided by the maximal length of hash sequence 1 and hash sequence 2 and the result is the syntactic similarity.

3.2.3 Semantic Similarity Analysis

The code semantics can also be used to measure the similarity of two code fragments, because the similarity of code fragments can be reflected from the word-choices in the source code. For example, if both two code fragments contain an identifier "login", they are probably related to the implementation of the login functionality. As a result, these two code fragments may be similar. Therefore, to further evaluate how similar two changed code fragments are, our approach analyzes the semantic similarity via analyzing the similarity of word-choices of two code fragments.

Algorithm 1. Sequence Matching Algorithm

```
Input:  $HashList1$: hash sequence 1;
        $HashList2$: hash sequence 2;
Output: $SyntaxSimilarity$
Begin
    //step 1: check the matched hashed in two sequences
1:    For $i = 0$ to $HashList1.length$ do:
2:      For $j = 0$ to $HashList2.length$ do:
3:        If ($HashList1$.get($i$) == $HashList2$.get($j$)) do:
4:          $Mtrx[i,j] = 1$; // two hashes are matched
5:        End If
6:      End For
7:    End For
    //step 2: find the subsequences from Mtrx
8:    Foreach $Mtrx[n,m]$ do:
9:      While (true) do:
10:       If($Mtrx[n , m] == 1$) do:
11:         $subseq_t$.add($Mtrx[n,m]$)
12:         $n$++;
13:         $m$++;
14:         $remove(Mtrx[n,m])$;
15:       Else:
16:         break;
17:       End If
18:     End While
19:   End Foreach
    //step 3: check the gap between subsequences
20:   Foreach $subseq_t$ do:
21:     While (true) do:
22:       If $gap(subseq_t, subseq_{t+1}) \leqslant 2$ do:// the gap is less
          than 3
23:         $subseq_t = link(subseq_t, subseq_{t+1})$; // link two
            subsequences
24:         $t = t + 1$;
25:         $remove(subseq_{t+1})$;
26:       Else:
27:         break;
28:       End If
29:     End While
30:   End Foreach
    //step 4: calculate the syntactic similarity
31:     $SyntaxSimilarity = \frac{max\_length\{subseq_1, subseq_2, \ldots, subseq_t\}}{max\_size\{HashList1, HashList2\}}$
32:   Return $SyntaxSimilarity$;
End
```

However, there may be some noise words that affect the semantic similarity analysis, e.g., variable "asass", "tttt", "kkk", and "b". These words may weaken the semantics of other words, and we need to filter them. To do that, a series of preprocessing rules are introduced: 1) spliting the camel-case words into single words, e.g., "setFamiyAddress" is split into "set", "famiy" and "address"; 2) filtering out the function words, such as "to", "the", "a", etc.; 3) filtering out the letters sequence which does not denote a word, such as "asass", "tttt"; 4) removing the punctuation and operational symbols from the source code. Additionally, we also apply the stem segmentation technique to help us identify the verbs in different tenses. Because English verbs may appear in different tenses, such as past tense, future tense, and perfect tense, we transform the verbs with different tenses into their original forms. This operation can help us accurately measure the similarity of two code fragments when they involve a verb with different tenses.

After the preprocessing, the code fragment is represented by a list of significative words. The preprocessing result of the code fragment is saved as a text document. However, the text document is not formed by meaningful sentences but a set of words. Therefore, we cannot directly employ the methods for calculating the semantic similarity of sentences, such as [53], to measure the semantic similarity of two code fragments. Instead, the vector space model[54] is introduced to build a term-document matrix for each code fragment, and then we calculate the semantic similarity of any two code fragments via the vetorial angle of their semantic vectors.

3.3 Mining Candidate Messages from Commit Repository

Given the changed code fragment of the target commit, our algorithm generates a hash sequence according to its code syntax, and extracts the significative words to generate the semantic vector. After that, we calculate the syntax similarity between the hash sequences and the semantic similarity between the semantic vectors of the target commit and the commit in local repository. To use a total similarity to represent the syntax and semantic similarities, we introduce (1).

$$CodeSimi = \alpha \times SyntSimi + \beta \times SemanSimi. \quad (1)$$

CodeSimi, *SyntSimi*, and *SemanSimi* are the total, syntax, and semantic similarity, respectively. α plus β equals 1.0. *CodeSimi* is a weighted value of *SyntSimi* and *SemanSimi*.

To measure the similarity of two commits, the first thought is to calculate the syntax and semantic similarities of their changed source code using (1). However, a commit consists of the code of current and previous versions. If the changed code fragments of two commits have a high similarity in their previous versions, i.e., before they are changed, we should reward their final similarity. When we apply the code change to two similar code fragments, and the modified code fragments are also similar, it is more possible that their code change targets are similar, and their messages are more likely to be interchangeable. The code fragments of the current version are used to calculate the total code similarity (*CodeSimi*), and the code fragments of the previous version are used to calculate the similarity of the code fragments before being changed. (2) shows how to calculate the similarity of the previous versions:

$$\begin{aligned} &PreCodeSimi \\ &= \alpha \times PreSyntSimi + \beta \times PreSemanSimi. \end{aligned} \quad (2)$$

Here, the values of α and β equal the ones in (1). Then, the final comprehensive commit similarity can be calculated by a weighted value of *CodeSimi* and *PreCodeSimi*, as (3) shows:

$$\begin{aligned} &ComprehSimi \\ &= \gamma \times CodeSimi + \delta \times PreCodeSimi. \end{aligned} \quad (3)$$

Here, $\gamma + \delta = 1.0$. For a target commit, our algorithm will calculate its *ComprehSimi* values with all the commits in the repository. The message of a commit with the highest *ComprehSimi* value will be recommended to the target commit if the *ComprehSimi* value is greater than a certain threshold θ.

4 Evaluation

4.1 Research Questions

In order to analyze ChangeDoc's capability to recommend commit messages, we would like to answer the following research questions in the evaluation.

*RQ*1. What is the performance of ChangeDoc in recommending commit messages?

*RQ*2. What are the reasons for a recommended message to be good or bad?

*RQ*3. For a fix message, what has to be done to fix it?

*RQ*4. How does ChangeDoc perform compared with existing work?

For a target commit, ChangeDoc retrieves commits in the repository, and ranks the searched results according to the comprehensive similarities. We find that if the comprehensive similarity is less than 0.4, it is difficult to find a reusable message for the target commit from the searched results. Therefore, to avoid recommending too many useless messages to users, we set the threshold $\theta = 0.4$ to filter out the low-similarity commits. To evaluate ChangeDoc in the experiment, we use the same weights parameters described in [20], i.e., $\alpha = 0.6$, $\beta = 0.4$, $\gamma = 0.8$, $\delta = 0.2$.

4.2 Dataset

Because ChangeDoc needs the existing commit messages for recommendation, we collect the projects from SourceForge② which have hundreds or thousands of commits (ranging from 500 to 30 000 commits). In total, we download 156 Java projects from SourceForge in our case study. However, not all the messages of commits are reusable due to the quality problem. Since ChangeDoc tries to recommend the messages of the existing commits to the target commit, we filter out the commits with no messages. Besides, as ChangeDoc needs to calculate the code similarity between commits, we filter out the commits without code change. Finally, there are more than half a million commits left after the filter. We build a local repository to save the commits, and our recommending algorithm works on the commit repository.

We apply ChangeDoc to recommend commit messages for 10 projects in the evaluation. The selected projects are commonly used in the evaluations of most of the software engineering related studies[45,55], they are: JHotDraw③, jEdit④, iText⑤, FreeCol⑥, Spring⑦, dcm4che⑧, openNMS⑨, UNICORE⑩, Makagiga⑪, and Kablink⑫. For the purpose of evaluating ChangeDoc on the projects coming from different domains, we select projects belonging to graphics editor, text editor, game, programming frame, network management software, middleware system, etc. The introduction of the projects is shown in Table 1.

Table 1. Projects Used in the Case Study

Project	Domain	Number of Commits
JHotDraw	Graphics	1 000
jEdit	Text	25 000
iText	Library	6 800
FreeCol	Game	9 600
dcm4che	Health	18 500
openNMS	Network	10 700
Spring	Frame	13 700
UNICORE	Middleware	20 000
Makagiga	Business	10 000
Kablink	Team software	23 000

4.3 Evaluation Criteria

Since the results of ChangeDoc require manual verification, we cannot verify all the messages of all commits of the evaluated projects. Instead, we choose 100 latest commits of each project to be used in message recommendation. We collect the 100 commits from the evolutionary history of each project along a backward time line. When we encounter the commits that have no message or the messages are less informative, we will filter out these commits until we have collected 100 commits. Note that the 156 projects in the local repository include these 10 projects. In our method, the message of the former commit is allowed to be recommended to the later commit of the same project (when the later commit is the target commit), but not vice versa.

We manually verify and evaluate the quality of the recommended commit messages. The recent work proposed by Wong *et al.*[31] evaluates the code messages based on the good, fix, and bad criteria. In this paper, we use the same criteria in evaluating the quality of recommended messages. We compare the original message of a commit with the recommended message, and manually verify if the recommended message satisfies the good\fix\bad criteria. If the meanings of the recom-

②https://sourceforge.net/, Apr. 2020
③http://www.jhotdraw.org, Feb. 2017.
④http://www.jedit.org, Sep. 2019.
⑤http://itextpdf.com, Oct. 2018.
⑥http://www.freecol.org, Oct. 2018.
⑦http://projects.spring.io/spring-framework/, Aug. 2019.
⑧http://www.dcm4che.org, Dec. 2019.
⑨https://www.opennms.org/en, May 2018.
⑩https://www.unicore.eu/, Apr. 2020.
⑪http://makagiga.sourceforge.net/, Mar. 2020.
⑫http://www.kablink.org/, Mar. 2020.

mended message and the original message are the same or similar, the recommended message is regarded as a good one. If the recommended message is not a good one but can express the same or similar meanings with the original message after minor modifications (e.g., replacing or deleting some words), it is regarded as a fix one. Otherwise, it is regarded as a bad one.

To make the verification more objective, three participants (i.e., one Ph.D. student and two postgraduates, all with no overlap with the authors of this paper) were invited to verify the results. In the verification, the first and the second participants manually evaluated the quality of all the recommended messages using good\fix\bad criteria. In this process, the two participants accomplished their work independently. If a result was given different criteria by the first two participants, the third participant would intervene to decide which result was correct.

4.4 Results Analysis

4.4.1 RQ1: Performance of ChangeDoc

To evaluate the performance of ChangeDoc, we apply ChangeDoc to 10 projects via using the original and improved similarity calculation algorithms, respectively (as mentioned in Subsection 3.2.1: the salient changed class identification). As Table 2 shows, we try to recommend messages for 1 000 commits from the 10 different projects. When setting $\theta = 0.4$, ChangeDoc recommends messages for 493 commits when employing the original similarity calculation algorithm (corresponding to the original algorithm of Table 2).

We perform a manual evaluation on all the recommended messages by the original similarity calculation algorithm, as shown in Table 2. We observe that 94 (19.1%) of the recommended messages are good, which can be directly used by the target commits. Up to 292 (59.2%) of the recommended messages are fix, which needs minor modifications, and 107 (21.7%) are bad.

Also in Table 2 (the improved algorithm), the results of ChangeDoc by employing the improved similarity calculation algorithm are listed. In general, the performance of ChangeDoc has been improved slightly by using the improved similarity calculation algorithm, and the commits successfully recommended messages by ChangeDoc reach 52.4%. We also observe that ChangeDoc with the improved algorithm provides an improvement from 19.1% to 21.5% for the good recommended commit messages, an improvement from 59.2% to 62.8% for the fix type of recommended commit messages, and a reduction from 21.7% to 15.7% for the bad ones.

The results show that ChangeDoc with the improved algorithm can recommend more good and fix messages and avoid more bad messages from the local repository. This indicates that using the changed fragments of the salient modified class can make the change intent of a commit more prominent, and further help to retrieve a suitable commit for the target one.

We also find that the good and fix cases from the improved algorithms cover the ones of the original algorithms. Namely, all the good and fix cases recommended by the original algorithms can be found in the results recommended by the improved algorithms. In addition, most of the bad cases recommended by the original algorithms and the improved algorithms are overlapped. This indicates that the improved algorithm keeps stable in the recommended results when

Table 2. Evaluation Results of ChangeDoc with Applying Original and Improved Algorithms

Project	Original Algorithm				Improved Algorithm			
	Good	Fix	Bad	Sum	Good	Fix	Bad	Sum
JHotDraw	8.0	27.0	13.0	48.0	9.0	28.0	11.0	48.0
jEdit	7.0	21.0	6.0	34.0	7.0	27.0	9.0	43.0
Spring	5.0	33.0	11.0	49.0	11.0	28.0	11.0	50.0
FreeCol	9.0	34.0	12.0	55.0	9.0	40.0	6.0	55.0
iText	4.0	30.0	9.0	43.0	5.0	38.0	5.0	48.0
dcm4che	18.0	20.0	8.0	46.0	20.0	37.0	2.0	59.0
openNMS	12.0	27.0	11.0	50.0	18.0	25.0	9.0	52.0
UNICORE	10.0	29.0	18.0	57.0	10.0	31.0	12.0	53.0
Makagiga	8.0	30.0	12.0	50.0	8.0	34.0	10.0	52.0
Kablink	13.0	41.0	7.0	61.0	15.0	41.0	8.0	64.0
SUM	94.0	292.0	107.0	493.0	112.0	329.0	83.0	524.0
Ratio (%)	19.1	59.2	21.7	49.3	21.5	62.8	15.7	52.4

compared with the original one.

Because we perform a manual verification on all the recommended messages, we analyze the reliability and validity of the manual verification. For the reliability analysis, the Cronbach's Alpha[56] is 0.92, which indicates the reliability of our manual verification is good. For the validity analysis, the KMO value[56] is 0.93, which indicates the validity of our manual verification is also very good.

Therefore, we answer RQ1 by concluding that ChangeDoc with the improved algorithm provides improvements to the good and fix recommended commit messages when compared with the original one, and ChangeDoc is able to recommend messages for 52.4% of the commits, in which there are 21.5% of good messages as well as 62.8% of fix messages.

4.4.2 *RQ2: Reasons Behind the Good and Bad Messages*

The result of RQ1 shows that 21.5% of the recommended messages are good messages. We classify the major reasons that make a message be good in Table 3.

Our results show that the most common reason for a good message is the sameness between the recommended messages and the original messages. Meanwhile, we notice that the number of good messages in project dcm4che is the largest, i.e., 20. After analyzing the evolution history of dcm4che, we can see that a bug can be fixed in different commits in different periods, and their messages are identical. As a result, if we use the latter released commit as a target commit, and the messages of previous released commits in the evolution will be recommended to it, then the recommended messages can be directly used by the target commit. We also count all the identical cases, and find that 57 identical recommended messages are from the previous commits of the same project and eight cases are from other projects.

The other two reasons for a good message are semantic inclusion and paraphrasing. Semantic inclusion means that the meaning of the recommended message can cover that of the original message, and they belong to an inclusion relationship. For example, as shown in Table 3, "performance optimization work" covers the meaning of "performance optimization for file syncbatch processing support". Paraphrasing means that two messages express the same meaning, but they use different words, for example, "it will require java 6" and "switched to Java 1.6", where java 6 is the same with Java 1.6.

In summary, we recommend messages for 524 commits in total, while 21.5% of the recommended messages can be directly used by the target commits. The majority of the good messages are due to the identical cases.

RQ1 shows that 15.7% of the recommended messages are bad ones. We also classify the major reasons that make a message not applicable in Table 4. The result shows that the most common reason for a bad message is that the message includes specific information. For example, "add toString method to BaseDocument" and "remove the EDIT button from Mac clients" are both meaningful messages. However, they are only

Table 3. Types of Good Messages

No.	Type	Description	Amount
1	Identical	The sentences of recommended and original messages are the same, e.g., two messages are: "consistent button size and positioning"	65 (58.0%)
2	Semantic inclusion	The meanings of recommended and original messages are of inclusion relation, e.g., original: "performance optimization for file syncbatch processing support"; recommended: "performance optimization work"	26 (23.2%)
3	Paraphrasing	The recommended and original messages have the same meaning but different expressions, e.g., original: "it will require java 6"; recommended: "switched to Java 1.6"	21 (18.8%)

Table 4. Types of Bad Messages

No.	Type	Description	Amount
1	Too specific	The recommended message contains too specific information to be suitable for more cases, e.g., "add toString method to Base-Document", "remove the EDIT button from Mac clients"	61 (74.5%)
2	Too much content	The recommended message contains too much information, and covers too broad scope, e.g., "Ported stable branch changes ... - enable running WebApp ... - added parms to web.xml ... - put static eventproxy ..."	22 (26.5%)

suitable for a specific context, and hard to be useful for other commits.

Another reason for a bad message is that the content of a message is complex. The example message shown in Table 4 (the second row) contains the information about several operations. The scope of this commit is too broad. As a result, its message is hard to be reused for other commits.

The majority of bad messages contain too specific information or too much content. However, it is reasonable. For a commit with a minor change, the change may be a specific code change in a specific project. For a commit with several changes for different purposes, its commit message is required to specify a list of changes.

4.4.3 RQ3: Reasons Behind the Fix Messages

The result of RQ1 shows that 62.8% of the recommended messages require minor modifications. We manually analyze these messages and classify the major reasons that make them repairable, as shown in Table 5.

In the results, the majority reason is the error objects (type 1). For example, the object of the original message "fix rollover button performance problem" is "rollover button", while the object of the recommended message "fix window scrolling performance problem" is "window scrolling". To make the recommended message reusable for the target commit, a correct object should be replaced.

The second main problem is that the recommended commit messages consist of incorrect sentence constituents (i.e., an adverbial or attributive error). As a result, only part of the sentences of these messages are reusable. For example, the recommended message describes "consistent button size and positioning for dialogs", while the original message has no adverbial constituent that is used to limit the "scope" (i.e., for dialogs). In this case, the direct-reuse of the recommended message becomes impossible.

The third reason for a fix message is due to the error subject (type 3), which is similar to the object error. For example, the original message "proper height default for JTable" is with the subject "JTable", while the recommended message "proper height default for JMenuTab" is with the subject "JMenuTab". A proper replacement for the subject can make the recommended message reusable.

Another problem is that the recommended messages may contain too many clauses (type 4), where most contents are suitable for describing the target commit, while some of the content may be irrelevant information. For example, the recommended message contains two clauses: "enable to configure maximal number of image compression" and "improve logging of concurrency". The message of the target commit is the same with the first clause, but the second clause is redundant. In such a case, the second clause is required to eliminate to obtain a reusable message.

Besides, in some cases, two of the fix types (type 5) may occur together in the same message. For example, a recommended message may contain both the object error and the subject error. It is worth noting that if a recommended message contains more than two fix types, we classify it as a bad one. The more fix types a message contains, the more difficult it is to make the message reusable in the target commit.

In summary, we conclude RQ3 that the majority of fix messages are due to the object error and incorrect sentence constituents. In fact, these problems can be repaired by leveraging natural language analyzing techniques to identify the consistency between messages and code changes in the commit. Inconsistency can possibly be solved by eliminating the redundancy or replacing object names.

Table 5. Types of Fix Messages

No.	Type	Description	Amount
1	Object error	The object of the recommended message is wrong, e.g., original: "fix rollover button performance problem"; recommended: "fix window scrolling performance problem"	105 (31.9%)
2	Half sentence reusable	The adverbial or attributive of the recommended message is wrong, e.g., original: "consistent button size and positioning"; recommended: "consistent button size and positioning for dialogs"	92 (27.9%)
3	Subject error	The subject of the recommended message is wrong, e.g., original: "proper height default for JTable"; recommended: "proper height default for JMenuTab"	58 (17.6%)
4	Partial clauses reusable	The recommended message contains multiple clauses, but only a part of them are needed, e.g., original: "enable to configure maximal number of image compression"; recommended: "enable to configure maximal number of image compression; improve logging of concurrency"	32 (9.7%)
5	Complex type	The combination of any two of above types, e.g., object error + subject error	42 (12.8%)

4.4.4 RQ4: Compared with Existing Work

1) *Compared with ChangeScribe*. In this subsection, we study the research question of what is the performance of ChangeDoc when compared it with ChangeScribe [9]. To address this question, we perform two empirical studies having different settings and involving different kinds of participants. Study 1 aims at assessing the conciseness, expressiveness and preciseness of the ChangeDoc commit messages with respect to those generated by ChangeScribe. Since the goal of study 1 does not require high experience or deep knowledge of the application domain, we involve mainly students. Study 2 aims at evaluating the importance of the items presented in the ChangeDoc messages, ChangeScribe messages, and original messages. In this case, the task assigned to participants is highly demanding; therefore we ask the experienced developers to evaluate the commit messages generated by ChangeDoc and ChangeScribe.

Conciseness, Expressiveness and Preciseness Evaluation. The goal of this study is to assess the conciseness, expressiveness and preciseness of commit messages generated by ChangeDoc and ChangeScribe. The context of this study consists of: objects, i.e., the messages generated by ChangeDoc and ChangeScribe from the 10 projects, and subjects evaluating the commit messages, i.e., 26 M.Sc. students, three Ph.D. students, and one faculty. All of these participants come from Sun Yat-sen University[13], and they have no overlap with the authors of this paper. We randomly select 60 of commits from the 10 subject projects, and use ChangeDoc and ChangeScribe to generate their messages.

We distribute the commits to the evaluators, in such a way that the 60 commits are evenly divided into three groups, and each group is evaluated by 10 participants (i.e., each participant is assigned 20 commits). We provide each participant with 1) a pre-study questionnaire; 2) the messages recommended by ChangeDoc; 3) the messages generated by ChangeScribe. Participants are asked to determine and indicate whether the content in the ChangeDoc recommended messages is 1) equally, 2) less , or 3) more concise (expressive, or precise), compared with that in the ChangeScribe generated messages. In order to avoid the bias in the evaluation, we do not tell participants by which method each commit message is generated. Specifically, in the questionnaires we show participants that the conciseness is used to measure whether the generated messages contain unnecessary information; expressiveness is used to evaluate whether the generated messages are readable and understandable; preciseness is used to assess whether the generated messages can accurately describe the code changes.

[13]http://www.sysu.edu.cn, Sept. 2020.

Table 6 summarizes the answers provided by the participants. On average, 94.67% of the messages generated by ChangeDoc are more concise than those generated by ChangeScribe, 4.83% equally concise, only 0.5% less concise. Because ChangeDoc directly utilizes the human-written messages, most of them contain less than 30 words in our statistics. The following is an example in which the content in the ChangeDoc message is more concise than that in the ChangeScribe message. The ChangeDoc message describes the implementation of NPE fixing: "Bug fixing: fixed possible NPE in getTargetStateId". In the ChangeScribe message, the change is reported as follows.

```
• BUG - FEATURE: <type-ID>
  ◦ This change set is mainly composed of:
   1.  Changes to package org.springframework.
      webflow.engine:
      Modifications to Transition.java:
      Add if statement at getTargetStateId() method;
      Add else part of (targetStateResolver!= null)
      condition;
      Add return statement at getTargetStateId().
```

Obviously, the ChangeDoc message describes the code change in a more concise manner.

Table 6. Conciseness, Expressiveness and Preciseness Evaluation

	Conciseness (%)			Expressiveness (%)			Preciseness (%)		
	Less	Same	More	Less	Same	More	Less	Same	More
Group 1	0.0	4.50	95.50	39.0	3.50	57.50	24.00	21.50	54.50
Group 2	1.0	1.00	98.00	28.5	7.00	64.50	42.00	11.00	57.00
Group 3	0.5	9.00	90.50	34.5	15.50	50.00	49.00	18.00	33.00
Average	0.5	4.83	94.67	34.0	8.67	57.33	38.33	16.83	44.83

On the other hand, 57.33% of the messages generated by ChangeDoc are more expressive than those generated by ChangeScribe, while 34% are less expressive. Because human-written messages usually describe the change intent behind a commit, ChangeDoc reusing the human-written messages can reflect the code changes more clearly. In the above example, this commit handles the null pointer exception. The ChangeDoc message describes the NPE fixing in a method. Although we can find the description of "Add else part of (targetStateResolver != null) condition" in the ChangeScribe message, we cannot realize this description corresponds to the NPE fixing if we do not analyze the source code of the commit.

At last, 44.83% of the messages generated by ChangeDoc are more precise than the messages generated by ChangeScribe, 38.33% are less precise, and 16.83% are equally precise. We observe that the handwritten messages by developers usually focus on the core change in a commit, but neglect the trivial changes in the same commit. On the contrary, ChangeScribe covers the code changes as many as possible in its generated message. As a result, a part of participants may consider that the ChangeScribe messages can describe the code changes in a more accurate manner, and they hold that 38.33% of the ChangeScribe messages are more precise than those generated by ChangeDoc.

Important Items Evaluation. The goal of this study is to evaluate the importance of the captured items in the tool-generated (i.e., ChangeDoc and ChangeScribe) and original commit messages from the perspective of software developers. The context of this study consists of: objects, i.e., 15 commits randomly selected from the 10 projects used in the case study, with ChangeDoc and ChangeScribe to automatically generate the messages of the selected commits, and subjects evaluating the messages, i.e., 10 professional software developers. We perform this study by questionnaire survey. The questionnaire consists of two parts on: 1) the participants' background and their experience in using and creating commit messages; 2) the evaluation of the ChangeDoc and ChangeScribe generated commit messages and the original ones.

The evaluation of each commit message is conducted as follows. Firstly, we point out to developers that some items (e.g., phrases) describe the meaning of a commit message. For example, the items containing in "Bug fixing: fixed possible NPE in getTargetStateId" are "Bug fixing", "fixed NPE" and "NPE in getTargetStateId". Then, participants are asked to indicate whether the item is: 1) not at all important; 2) unimportant; 3) important; or 4) very important. We hope that developers can vote to select an item that is most important to represent the meaning of a message.

The 10 evaluators are professional developers, who report experience in software development ranging from 3 to 11 years (median 6). Also, four out of the 10 evaluators declare that they use commit messages frequently (i.e., more than 10 times per month), mainly to check for bug fixes in a software system. The other six developers declare that they occasionally check in the commit messages for peer code review to deal with the code consistency and compatibility issues. In addition, most of the evaluators report they have created commit messages many times.

Going to the core of this study, the answers provided by the 10 developers on the importance of the terms from the tool-generated (i.e., ChangeDoc and ChangeScribe) and the original messages are summarized in Figs.5–7, respectively. Among the items presented in the messages generated by ChangeDoc, the ones considered as important/very important by developers are: "Performance optimization", "Server not started", "Handle infinite loop", "Allow sharing", etc., summarizing the most important changes in the commit. On the other hand, for the items presented in the ChangeScribe generated messages the ones considered as important/very important by developers are: "Add else part", "Modify conditional expression 2", "Modify arguments", "Remove variable". We can observe that the items presented in the messages generated by ChangeDoc usually describe the change intent of a commit, i.e., these terms indicate "why" the change occurs. In contrast, the items presented in the messages gene-

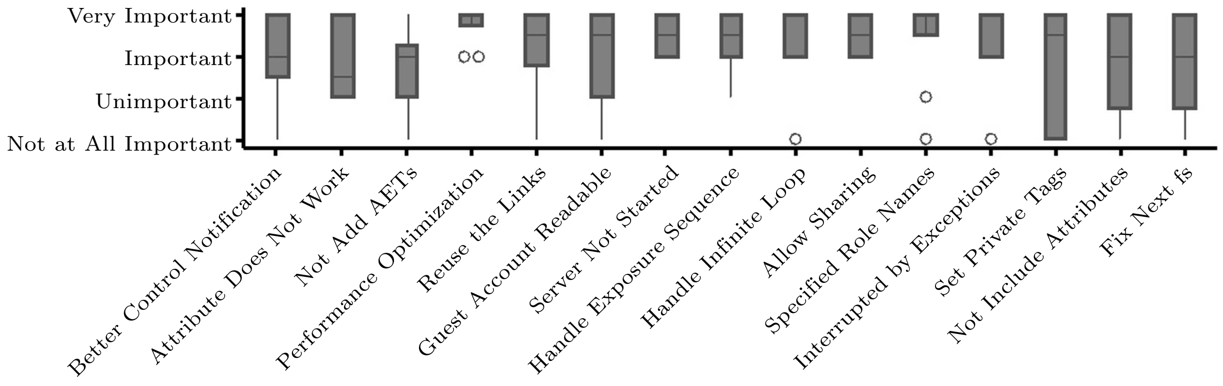


Fig.5. Important terms reported by the evaluators for the commit messages generated by ChangeDoc.

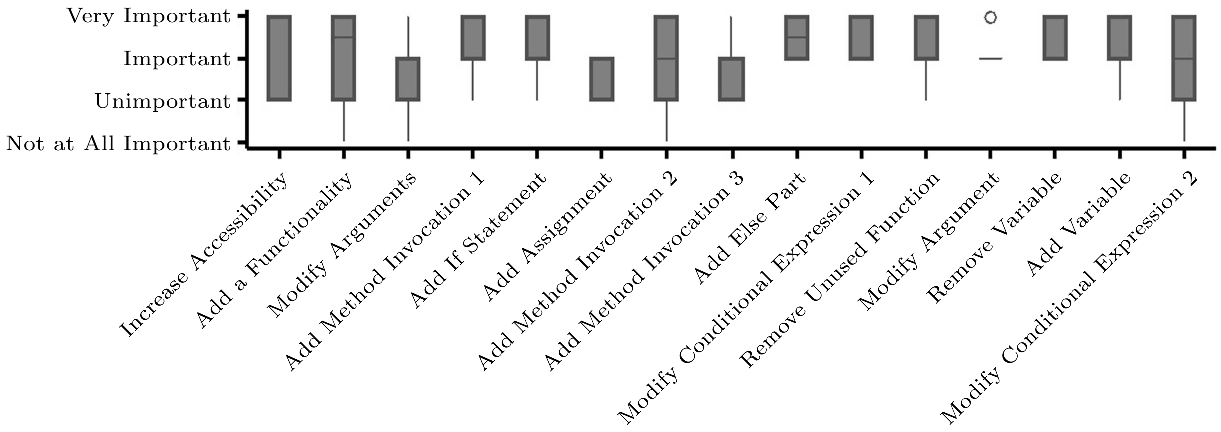


Fig.6. Important terms reported by the evaluators for the commit messages generated by ChangeScribe.

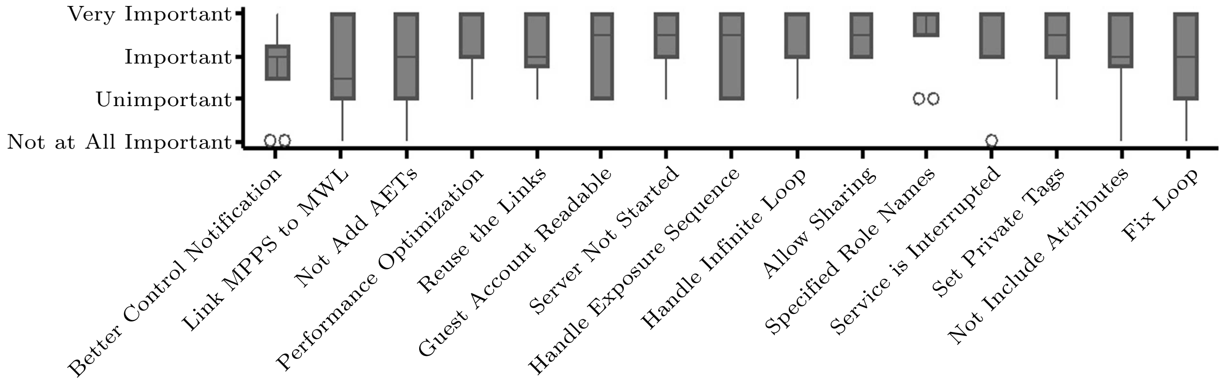


Fig.7. Important terms reported by the evaluators for the original commit messages.

rated by ChangeScribe describe the code changes at a fine-grained level, and usually describe "what" has been changed in a commit. Therefore, the terms presented in Fig.5 confirm that our method can generate the change intent behind the code changes.

On the other hand, we can also observe that most of the terms in the messages generated by ChangeDoc are presented in the original commit messages (see Fig.7). For example, 12 out of the 15 messages generated by ChangeDoc are with the same terms to the original messages. Because the terms selected from original messages can be regarded as the ones that can best represent the meaning of the messages, the messages generated by ChangeDoc can capture most of the important information of the code change.

In summary, ChangeScribe messages usually describe the "what" information of code changes, while ChangeDoc messages describe the "why" information of code changes. Also, most terms considered as important or very important in the original messages can be captured by ChangeDoc messages.

There are some guidances that might be given to the developers from the results. First, the terms in the original messages focus more on a specific task, such as "Server not started", "Handle infinite loop", and this suggests that developers put more description on a specific task when they write the commit messages. Second, the original messages usually describe the "why" information of code changes, rather than the "what" information of code changes. This suggests that developers focus more on the intent of a code change when they write the commit messages.

2) *Compared with NNGen.* We have compared ChangeDoc with the updated work NNGen[26]. To run NNGen, we use the model proposed by Liu *et al.*[26] Then, we use the 1 000 commits (coming from the 10 projects) used in the experiment of this paper as the test set. NNGen can generate messages for every commit. For the sake of fairness, we set the value of threshold θ as 0, and then ChangeDoc can generate messages for every commit by recommending the message of a commit with the highest ComprehSimi value.

The average BLEU values for NNGen and ChangeDoc can be observed in Table 7. We can see that the performances of ChangeDoc and NNGen are almost on the same level. In essence, ChangeDoc and NNGen have similar principles, and both of them seek a similar commit from the historical commits, and recommend the message of a commit with the highest similarity to the target commit. NNGen employs the nearest neighbor algorithm with bags-of-words to find the similar commit, while ChangeDoc uses the clone algorithm and the syntactic and semantic information to find the similar commit. Although the methods used are different, they are both effective in finding similar commits from historical commits.

Table 7. Comparing ChangeDoc with NNGen

Approach	BLEU-1	BLEU-2	BLEU-3	BLEU-4
NNGen	25.52	18.76	16.44	13.79
ChangeDoc	25.68	19.26	16.67	14.54

5 Discussion: θ Choice Effect

In this paper, the threshold θ is set to avoid recommending too many useless messages to users. Namely, when the highest *ComprehSimi* value is greater than threshold θ, and the message of a commit will be recommended to the target commit. We have analyzed the impact of the threshold choice on the performance of ChangeDoc, as Table 8 shows.

Table 8. Threshold Choice Effect on the Performance of ChangeDoc

No.	Threshold Value	Covered Commit (%)	Good (%)	Fix (%)
1	0.2	63.6	16.3	51.5
2	0.3	58.5	19.5	55.4
3	0.4	52.4	21.5	62.8
4	0.5	33.5	22.6	64.1
5	0.6	18.3	24.1	65.4

When the threshold value is 0.4, ChangeDoc is able to recommend messages for 52.4% of the commits (i.e., covered commits in Table 8) with a performance of 21.5% good cases and 62.8% fix cases. When the threshold value is 0.2 or 0.3, ChangeDoc is able to recommend more messages for the commits, but the performance of recommending good and bad cases becomes worse. The result indicates that ChangeDoc can recommend messages for more commits, but the quality of recommended messages has declined. Similarly, when the threshold value is 0.5 or 0.6, although the proportion of recommending good and bad cases increases, the proportion of the covered commits decreases seriously. As a compromise, we set the threshold value as 0.4 for ChangeDoc.

6 Threats to Validity

The main threat to external validity is the suitability of our evaluation measure. We use a new measure to evaluate the effectiveness of the proposed approach in this paper. We compare the original message of a commit with the recommended messages, and manually verify if the recommended message satisfies the good\fix\bad criteria. In the manual verification, we invite three participants to verify the quality of recommended messages. Two participants manually verify the quality of all the recommended messages independently, and the other one participant will intervene if there exist conflicts in the verified results. While, even with these restricted conditions, it is still possible that a small portion of results are misjudged by the participants. In the future, we need to invite more participants to mitigate this threat.

A threat to external validity is the generalizability of our results. We have recommended messages for 1 000 commits from 10 different open-source Java software projects. When applying our approach to projects written by other programming languages, some particular code syntax (e.g., pointer operation in C/C++) should be carefully handled when extracting the code syntax. In the future, further investigation by analyzing even more projects written by other programming languages is needed to mitigate this threat.

To compare ChangeDoc with ChangeScribe, we perform two empirical studies having different settings and involving different kinds of participants in RQ4. The message generated by ChangeScribe follow some patterns and may be easy to recognize by the participants. Then, this may cause some interference in the evaluation. In the future, inviting more participants to involve in the empirical studies can mitigate this threat. Because fewer validation tasks are assigned to single participant, the participant is not easy to recognize messages generated by ChangeScribe.

In this work, we have taken some measures to disambiguate the meaning of words. For example, an English verb may appear in different tenses, such as past tense, future tense, and perfect tense, and we transform verbs of different tenses into their original forms. Despite these measures, it is possible that the problem of Word Sense Disambiguation (i.e., WSD) [57] may not be completely avoided. In the future, some more effective WSD techniques are needed to mitigate this threat.

7 Conclusions

In this paper, we proposed ChangeDoc to automatically recommend messages for the target commits by mining version control systems. Given a target commit, ChangeDoc retrieves a most similar commit in the local repository from the code syntactic and semantic perspectives, and the message of the most similar commit is recommended to the target commit. The experimental results showed that 21.5% of the recommended messages are good, 62.8% are fix, and 15.7% are bad. Besides, we analyzed the reasons which make a message reusable or non-reusable in detail, and we also compared ChangeDoc with ChangeScribe in terms of conciseness, expressiveness, preciseness and the important terms.

In the future, we plan to evaluate ChangeDoc with more commits from more software projects and develop a better technique which could improve the accuracy of commit message recommendation further.

References

[1] Barnett M, Bird C, Brunet J, Lahiri S K. Helping developers help themselves: Automatic decomposition of code review changesets. In *Proc. the 37th IEEE/ACM International Conference on Software Engineering*, May 2015, pp.134-144.

[2] Huang Y, Jia N, Zhou Q, Chen X, Xiong Y F, Luo X N. Guiding developers to make informative commenting decisions in source code. In *Proc. the 40th IEEE/ACM International Conference on Software Engineering: Companion*, May 2018, pp.260-261.

[3] Hattori L, Lanza M. On the nature of commits. In *Proc. the 23rd IEEE/ACM International Conference on Automated Software Engineering*, September 2008, pp.63-71.

[4] Huang Y, Huang S, Chen H, Chen X, Zheng Z, Luo X, Jia N, Hu X, Zhou X. Towards automatically generating block comments for code snippets. *Information and Software Technology*, 2020, 127: Article No. 106373.

[5] Tao Y, Dang Y, Xie T, Zhang D, Kim S. How do software engineers understand code changes? An exploratory study in industry. In *Proc. the 20th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, November 2012, Article No. 51.

[6] Huang Y, Chen X, Zou Q, Luo X. A probabilistic neural network-based approach for related software changes detection. In *Proc. the 21st Asia-Pacific Software Engineering Conference*, Dec. 2014, pp.279-286.

[7] Maalej W, Happel H J. Can development work describe itself? In *Proc. the 7th International Working Conference on Mining Software Repositories*, May 2010, pp.191-200.

[8] Dyer R, Nguyen H A, Rajan H, Nguyen T N. Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In *Proc. the 35th International Conference on Software Engineering*, May 2013, pp.422-431.

[9] Linares-Vásquez M, Cortés-Coy L F, Aponte J, Poshyvanyk D. ChangeScribe: A tool for automatically generating commit messages. In *Proc. the 37th IEEE/ACM International Conference on Software Engineering*, May 2015, pp.709-712.

[10] Moreno L, Bavota G, Penta M D, Oliveto R, Marcus A, Canfora G. Automatic generation of release notes. In *Proc. the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, November 2014, pp.484-495.

[11] Moreno L, Bavota G, Penta M D, Oliveto R, Marcus A, Canfora G. ARENA: An approach for the automated generation of release notes. *IEEE Transactions on Software Engineering*, 2016, 43(2): 106-127.

[12] Shen J, Sun X, Li B, Yang H, Hu J. On automatic summarization of what and why information in source code changes. In *Proc. the 40th IEEE Annual Computer Software and Applications Conference*, June 2016, pp.103-112.

[13] Buse R P, Weimer W R. Automatically documenting program changes. In *Proc. the 25th IEEE/ACM International Conference on Automated Software Engineering*, September 2010, pp.33-42.

[14] Rastkar S, Murphy G C. Why did this code change? In *Proc. the 35th International Conference on Software Engineering*, May 2013, pp.1193-1196.

[15] Parnin C, Görg C. Improving change descriptions with change contexts. In *Proc. the 2008 International Working Conference on Mining Software Repositories*, May 2008, pp.51-60.

[16] Sridhara G, Hill E, Muppaneni D, Pollock L, Vijay-Shanker K. Towards automatically generating summary comments for Java methods. In *Proc. the 25th IEEE/ACM International Conference on Automated Software Engineering*, September 2010, pp.43-52.

[17] Moreno L, Aponte J, Sridhara G, Marcus A, Pollock L, Vijay-Shanker K. Automatic generation of natural language summaries for Java classes. In *Proc. the 21st International Conference on Program Comprehension*, May 2013, pp.23-32.

[18] Spinellis D. Version control systems. *IEEE Software*, 2005, 22(5): 108-109.

[19] Zhong H, Meng N. Towards reusing hints from past fixes: An exploratory study on thousands of real samples. In *Proc. the 40th IEEE/ACM International Conference on Software Engineering*, May 2018, pp.885-885.

[20] Huang Y, Zheng Q, Chen X, Xiong Y, Liu Z, Luo X. Mining version control system for automatically generating commit comment. In *Proc. the 2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, November 2017, pp.414-423.

[21] Cortes-Coy L F, Linares-Vásquez M, Aponte J, Poshyvanyk D. On automatically generating commit messages via summarization of source code changes. In *Proc. the 14th IEEE International Working Conference on Source Code Analysis and Manipulation*, September 2014, pp.275-284.

[22] Jiang S, McMillan C. Towards automatic generation of short summaries of commits. arXiv:1703.09603, 2017. https://arxiv.org/abs/1703.09603, Sept. 2020.

[23] Jiang S, Armaly A. Automatically generating commit messages from diffs using neural machine translation. In *Proc. the 32nd IEEE/ACM International Conference on Automated Software Engineering*, October 2017, pp.135-146.

[24] Hoang T, Kang H J, Lawall J, Lo D. CC2Vec: Distributed representations of code changes. arXiv:2003.05620, 2003. https://arxiv.org/pdf/2003.05620.pdf, Sept. 2020.

[25] Xu S, Yao Y, Xu F, Gu T, Tong H, Lu J. Commit message generation for source code changes. In *Proc. the 28th International Joint Conference on Artificial Intelligence*, August 2019, pp.3975-3981.

[26] Liu Z, Xia X, Hassan A E, Lo D, Xing Z, Wang X. Neural-machine-translation-based commit message generation: How far are we? In *Proc. the 33rd ACM/IEEE International Conference on Automated Software Engineering*, September 2018, pp. 373-384.

[27] Nie L Y, Gao C, Zhong Z, Lam W, Liu Y, Xu Z. Contextualized code representation learning for commit message generation. arXiv:2007.06934, 2020. https://arxiv.org/pdf/2007.06934, Sept. 2020.

[28] Liu S, Gao C, Chen S, Nie L Y, Liu Y. ATOM: Commit message generation based on abstract syntax tree and hybrid ranking. arXiv:1912.02972, 2019. https://arxiv.org/abs/1912.02972, Sept. 2020.

[29] McBurney P W, McMillan C. Automatic documentation generation via source code summarization of method context. In *Proc. the 22nd International Conference on Program Comprehension*, June 2014, pp.279-290.

[30] Wong E, Yang J, Tan L. AutoComment: Mining question and answer sites for automatic comment generation. In *Proc. the 28th IEEE/ACM International Conference on Automated Software Engineering*, November 2013, pp.562-567.

[31] Wong E, Liu T, Tan L. CloCom: Mining existing source code for automatic comment generation. In *Proc. the 22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering*, March 2015, pp.380-389.

[32] Haiduc S, Aponte J, Moreno L, Marcus A. On the use of automated text summarization techniques for summarizing source code. In *Proc. the 17th Working Conference on Reverse Engineering*, October 2010, pp.35-44.

[33] Haiduc S, Aponte J, Marcus A. Supporting program comprehension with source code summarization. In *Proc. the 32nd ACM/IEEE International Conference on Software Engineering*, May 2010, pp.223-226.

[34] Iyer S, Konstas I, Cheung A, Zettlemoyer L. Summarizing source code using a neural attention model. In *Proc. the 54th Annual Meeting of the Association for Computational Linguistics*, August 2016, pp.2073-2083.

[35] Allamanis M, Peng H, Sutton C. A convolutional attention network for extreme summarization of source code. In *Proc. the 33rd International Conference on Machine Learning*, June 2016, pp.2091-2100.

[36] Hu X, Li G, Xia X, Lo D, Jin Z. Deep code comment generation. In *Proc. the 26th IEEE International Conference on Program Comprehension*, May 2018, pp.200-210.

[37] Hu X, Li G, Xia X, Lo D, Lu S, Jin Z. Summarizing source code with transferred API knowledge. In *Proc. the 27th International Joint Conference on Artificial Intelligence*, July 2018, pp.2269-2275.

[38] Baxter I D, Yahin A, de Moura L M *et al.* Clone detection using abstract syntax trees. In *Proc. the 1998 Int. Conf. Software Maintenance*, November 1998, pp.368-377.

[39] Roy C K, Cordy J R, Koschke R. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming*, 2009, 74(7): 470-495.

[40] Wettel R, Marinescu R. Archeology of code duplication: Recovering duplication chains from small duplication fragments. In *Proc. the 7th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, September 2005, pp.63-70.

[41] Yuan Y, Guo Y. Boreas: An accurate and scalable token-based approach to code clone detection. In *Proc. the 27th IEEE/ACM International Conference on Automated Software Engineering*, Sept. 2012, pp.286-289.

[42] Kamiya T, Kusumoto S, Inoue K. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 2002, 28(7): 654-670.

[43] Fluri B, Wuersch M, PInzger M, Gall H. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Transactions on Software Engineering*, 2007, 33(11): 725-743.

[44] Misra J, Annervaz K, Kaulgud V. Software clustering: Unifying syntactic and semantic features. In *Proc. the 19th Working Conference on Reverse Engineering*, October 2012, pp.113-122.

[45] Huang Y, Chen X, Liu Z, Luo X, Zheng Z. Using discriminative feature in software entities for relevance identification of code changes. *Journal of Software: Evolution and Process*, 2017, 29(7): Article No. 2.

[46] Huang Y, Jia N, Chen X, Hong K, Zheng Z. Salient-class location: Help developers understand code change in code review. In *Proc. the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, November 2018, pp.770-774.

[47] Khatchadourian R, Rashid A, Masuhara H, Watanabe T. Detecting broken pointcuts using structural commonality and degree of interest (N). In *Proc. the 30th IEEE/ACM International Conference on Automated Software Engineering*, Nov. 2015, pp.641-646.

[48] Nguyen H A, Nguyen A T, Nguyen T T, Nguyen T N, Rajan H. A study of repetitiveness of code changes in software evolution. In *Proc. the 28th IEEE/ACM International Conference on Automated Software Engineering*, Nov. 2013, pp.180-190.

[49] Gao Q, Zhang H, Wang J, Xiong Y, Zhang L, Mei H. Fixing recurring crash bugs via analyzing Q & A sites (T). In *Proc. the 30th IEEE/ACM International Conference on Automated Software Engineering*, Nov. 2015, pp.307-318.

[50] Huang Y, Hu X, Jia N, Chen X, Xiong Y, Zheng Z. Learning code context information to predict comment locations. *IEEE Transactions on Reliability*, 2020, 69(1): 88-105.

[51] Huang Y, Jia N, Shu J, Hu X, Chen X, Zhou Q. Does your code need comment? *Software — Practice and Experience*, 2020, 50(3): 227-245.

[52] Huang Y, Hu X, Jia N, Chen X, Zheng Z, Luo X. CommtPst: Deep learning source code for commenting positions prediction. *Journal of Systems and Software*, 2020, 170: Article No. 110754.

[53] Oliva J, Serrano J I, del Castillo M D, Iglesias Á. SyMSS: A syntax-based measure for short-text semantic similarity. *Data & Knowledge Engineering*, 2011, 70(4): 390-405.

[54] Salton G. A vector space model for automatic indexing. *Communications of the ACM*, 1975, 18(11): 613-620.

[55] Zhang J, Chen J, Hao D, Xiong Y, Xie B, Zhang L, Mei H. Search-based inference of polynomial metamorphic relations. In *Proc. the 2014 ACM/IEEE International Conference on Automated Software Engineering*, September 2014, pp.701-712.

[56] Li Q. A novel Likert scale based on fuzzy sets theory. *Expert Systems with Applications*, 2013, 40(5): 1609-1618.

[57] Navigli R. Word sense disambiguation: A survey. *ACM Computing Surveys*, 2009, 41(2): 115-183.

Yuan Huang received his Ph.D. degree in computer science from Sun Yat-sen University, Guangzhou, in 2017. He is an associate research fellow in the School of Data and Computer Science, Sun Yat-sen University, Guangzhou. He is particularly interested in software evolution and maintenance, code analysis and comprehension, and mining software repositories.

Nan Jia received her Ph.D. degree in computer science from Sun Yat-sen University, Guangzhou, in 2017. She is an associate professor at the School of Information Engineering, Hebei GEO University, Shijiazhuang. She is particularly interested in data mining and software engineering.

Hao-Jie Zhou is a postgraduate student at the Sun Yat-sen University, Guangzhou. His research interest includes software engineering, code analysis and comprehension, and mining software repositories.

Xiang-Ping Chen is an associate professor at the Sun Yat-sen University, Guangzhou. She got her Ph.D. degree in software engineering from the Peking University, Beijing, in 2010. Her research interest includes software engineering and mining software repositories.

Zi-Bin Zheng received his Ph.D. degree from the Chinese University of Hong Kong, Hong Kong, in 2011. He is currently a professor at School of Data and Computer Science with Sun Yat-sen University, Guangzhou. He published over 120 international journal and conference papers, including three ESI highly-cited papers. His research interests include blockchain, services computing, software engineering, and financial big data. He was a recipient of several awards, including the Top 50 Influential Papers in Blockchain of 2018, the ACM SIGSOFT Distinguished Paper Award at ICSE 2010, and the Best Student Paper Award at ICWS 2010.

Ming-Dong Tang is a professor in the School of Information Science and Technology at Guangdong University of Foreign Studies, Guangzhou. He received his Ph.D. degree in computer science from the Institute of Computing Technology, Chinese Academy of Sciences, Beijing, in 2010. His current research interests include services computing, cloud computing, and social networks. He is a member of ACM and IEEE.