The Journal of Systems and Software 153 (2019) 45–63

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Automatically detecting the scopes of source code comments

Huanchao Chen[a], Yuan Huang[a], Zhiyong Liu[a], Xiangping Chen[b,*], Fan Zhou[a], Xiaonan Luo[c]

[a] National Engineering Research Center of Digital Life, School of Data and Computer Science, Sun Yat-sen University, Guangzhou, China
[b] Guangdong Key Laboratory for Big Data Analysis and Simulation of Public Opinion, the School of Communication and Design, Sun Yat-sen University, Guangzhou, China
[c] School of Computer Science and Information Security, Guilin University of Electronic Technology, Guilin, China

ARTICLE INFO

Article history:
Received 25 August 2018
Revised 29 January 2019
Accepted 11 March 2019
Available online 12 March 2019

Keywords:
Comment scope detection
Machine learning
Software repositories

ABSTRACT

Comments convey useful information about the system functionalities and many methods for software engineering tasks take comments as an important source for many software engineering tasks such as code semantic analysis, code reuse and so on. However, unlike structural doc comments, it is challenging to identify the relationship between the functional semantics of the code and its corresponding textual descriptions nested inside the code and apply it to automatic analyzing and mining approaches in software engineering tasks efficiently.

In this paper, we propose a general method for the detection of source code comment scopes. Based on machine learning, our method utilized features of code snippets and comments to detect the scopes of source code comments automatically in Java programs. On the dataset of comment-statement pairs from 4 popular open source projects, our method achieved a high accuracy of 81.45% in detecting the scopes of comments. Furthermore, the results demonstrated the feasibility and effectiveness of our comment scope detection method on new projects.

Moreover, our method was applied to two specific software engineering tasks in our studies: analyzing software repositories for outdated comment detection and mining software repositories for comment generation. As a general approach, our method provided a solution to comment-code mapping. It improved the performance of baseline methods in both tasks, which demonstrated that our method is conducive to automatic analyzing and mining approaches on software repositories.

1. Introduction

Comments providing insight for code functionalities are widely used to assist developers in software comprehension and maintenance (Woodfield et al., 1981; Ko et al., 2006; Aggarwal et al., 2002). As the most-used documentary artifact for code understanding besides the code, comments are used as standard practice in software development to increase the readability of code and express programmers intentions in a more explicit manner (de Souza et al., 2005; Huang et al., 2018a).

The scope of a comment means a region where the comment refers to in the program. It contains a few statements that match the description or implement the functionality mentioned in the comment. There are three types of comments: doc comments, block comments and line comments. Doc comments, known as Javadocs in Java, are directly associated with classes, methods, or attributes, whereas block and line comments are nested inside the code (Ambler et al., 1999). Thus, the scope of a doc comment is the whole method obviously, and the scope of a block comment or a line comment is not clear.

Comments convey useful information about the system functionalities, so many approaches for software engineering tasks take comments as an important source for code semantic analysis. To the best of our knowledge, detecting the scopes of comments can contribute to the following software engineering tasks:

- Analyzing Software repositories. Many approaches were proposed to analyze software repositories to assist in program comprehension and software evolution tasks (Fluri et al., 2007; Tan et al., 2012; Fluri et al., 2009; DAmbros et al., 2008; Thomas et al., 2014; Huang et al., 2017a; Huang et al., 2018b). When investigated the co-evolution relation between source code and its associated comments in software repositories, we have to map source code entities to comments in the code (Fluri et al.,

* Corresponding author.
E-mail address: chenxp8@mail.sysu.edu.cn (X. Chen).

https://doi.org/10.1016/j.jss.2019.03.010

```
@Override
public boolean insert(View view, Buffer buffer, String path){
      File file = new File(path);

       //{{{ Check if file is valid
       if(!file.exists())
             return false;

       if(file.isDirectory()){
             VFSManager.error(view,file.getPath(),"ioerror.open-directory",null);
             return false;
       }

       if(!file.canRead()){
             VFSManager.error(view,file.getPath(),"ioerror.no-read",null);
             return false;
       } //}}}
    ...
}
```

Fig. 1. A code snippet of file FileVFS.java from jEdit.

2007; Tan et al., 2012; Fluri et al., 2009). Detecting the scopes of comments is a straight-forward method to track relations between comments and source code entities algorithmically. After relating comments to code snippets, we could investigate whether or not comments are adapted when source code changes, and check for comment-code inconsistencies as well.

- Mining software repositories. With the development of information retrieval approaches for software repository mining tasks including concept/feature/concern location, impact analysis, code retrieval and reuse (Haiduc et al., 2013; Marcus and Antoniol, 2012) and so on, comments are widely used in narrowing the lexical gap between the natural language used in the user query and the programming language. Detecting the scopes of comments is the basis for many mining tasks. For example, from comment-code mappings, many automatic techniques can mined semantically similar words in the software domain (Howard et al., 2013; Yang and Tan, 2012). Moreover, during code comment generation, it is essential to determine the scopes of comments when selecting related comments to the cloned code snippets (Wong et al., 2015) or building the dataset of comments and code pairs for code to comment translation (Zheng et al., 2017).

However, most automatic methods for software engineering tasks only utilize doc comments (Tan et al., 2012; Howard et al., 2013; Yang and Tan, 2012; Tan et al., 2007) because doc comments are directly mapped to code snippets and the scopes of block/line comments that have no specifications are difficult to determine.

Some works noticed the importance of detecting the scope of block/line comments. In order to utilize block/line comments efficiently, Kaelbling proposed to use scoped comments instead of block/line comments (Kaelbling, 1988). In practice, some block/line comments' scopes are explicitly marked in widely known open source projects such as jEdit[1] and Hibernate.[2] Fig. 1 and Fig. 2 illustrate two examples. As shown in Fig. 1, the authors of jEdit introduced two special markers: "//{{{" and "//}}}" to indicate the beginning and the end of the scope of the leading comment *"Check if file is valid"*. Similarly, in the source code of Hibernate illustrated in Fig. 2, two special marks "//*tag* :: " and "//*end* :: " are used to indicate the beginning and the end of the scope of the comment respectively. In this way, the markers are helpful for detecting the scopes of comments automatically by simple text manipulation. However, marking the scope of comment is time consuming and trivial for developers. Most block/line comments' scopes are not marked in existing projects.

In order to detect the scope of block/line comments automatically, some works proposed to use textual similarities calculation and heuristic rules to deal with the problem of the association between the comments and code (McBurney and McMillan, 2016; Wong et al., 2015; Fluri et al., 2007). However, detecting relations between comments and source code entities is complicated and difficult that the accuracy of existing detecting methods is unsatisfied (Liu et al., 2018; Wong et al., 2015). For example, the experiment result of comment recommendation method showed that the increase of scope detection accuracy can improve the recommendation efficiency (Wong et al., 2015).

For efficient use of block/line comments in source code analysis, in this paper, we propose a general approach to determine the scopes of source code comments automatically by machine learning. Specifically, we designed and implemented our method that utilizes random forests to detect comments's scopes. It contains three phases: given the source code of Java projects, our method first extracts block and line comments within the method bodies of the source code. Then it segments the source code into comment and code pairs by heuristic rules and filters out inappropriate comments for analysis. In the second phase, taking a statement as an analysis unit, our method extracts three dimensions of features including comment features, code features and comment-code relationship features from the statement and its corresponding comment. In the third phase, utilizing machine learning, our method classifies the statements into two categories: within and outside the comment scopes. Based on the categorized results, our method finally applies a strategy to determine the scopes of code comments automatically.

To evaluate our comment scope detection method, we built a dataset consisting of 4000 comment-statement pairs. The dataset was manually validated through a webpage. We evaluated the accuracy of our method on the validated dataset of comment-statement pairs from 4 popular open source projects. The result showed that our method achieved a high accuracy of 81.45% in detecting the scopes of comments. We also investigated the effect of the sizes of comment scopes on the accuracy of our method and found that our method performs better when the scopes of com-

[1] http://www.jedit.org/.
[2] http://hibernate.org/.

```
@Test
public void testFlushAutoCommit() {
  EntityManager entityManager = null;
  EntityTransaction txn = null;
  try {
      //tag::flushing-auto-flush-commit-example[]
      entityManager = entityManagerFactory().
          createEntityManager();
      txn = entityManager.getTransaction();
      txn.begin();

      Person person = new Person( "John Doe" );
      entityManager.persist( person );
      log.info( "Entity is in persisted state" );

      txn.commit();
      //end::flushing-auto-flush-commit-example[]
  ...
  }
}
```

Fig. 2. A code snippet of file SpatialTest.java from Hibernate.

ments cover a small range of statements. Furthermore, the experimental results demonstrated the feasibility and effectiveness of our comment scope detection method when applying to a brand new project.

More generally, our method was applied to two software engineering tasks in our studies: analyzing software repositories for outdated comment detection and mining software repositories for comment generation. As a general approach, our method provided a viable solution to comment-code mapping. It improved the performance and efficiency of baseline methods in both tasks, which demonstrated that our method is conducive to automatic approaches on analyzing and mining software repositories in software engineering.

In summary, this paper makes the following contributions.

- We are the first to design and implement a general method, utilizing machine learning, for the automatic detection of scopes of block/line comments whose scopes have no specifications. Our method takes source code files as input and no other information is required.
- With the validated dataset, we conducted a comprehensive evaluation of our approach. Our evaluation showed that our method outperformed heuristic rules, and worked stably well even in new projects without any prior knowledge.
- As a general method, we applied our method to assist two normal software engineering tasks: outdated comment detection and automatic comment generation. Our method improve the performance of the baseline methods (Liu et al., 2018; Wong et al., 2015) in both two tasks. The studies reveals that our method can be used to assist automatic approaches in software engineering.
- We created the dataset that can be used as benchmarks for comment scope detection. The dataset has manually labeled categories. It consists of 4000 comment-statement pairs belonging to 4 open source projects and implemented in Java.

We significantly extended our previous work (Chen et al., 2018). In particular, we converted the heuristic rules into an algorithm and made a detailed description about the process of source code segmentation. The data set has been doubled to 4000 comments to reduce the risk of overfitting. Moreover, we added *f-score* metric to better evaluate the effectiveness of our method in our experiments. Last but no least, we applied our method to two previous approaches for two software engineering tasks and compared our methods against them in additional studies.

In the following part of this paper, we will first introduce background knowledge in Section 2, including word embeddings and random forests. In Section 3, we will discuss our approach in detail. Section 4 will expound on all our experiments to evaluate our method's capabilities to detect the scopes of source code comments and describe applications of our method to two software engineering tasks. Section 5 presents related work. Section 6 explains threats to validity, and Section 7 concludes the paper.

2. Background

This section introduces the two techniques used in our approach: word embeddings and random forests.

2.1. Word embeddings

Word embeddings are unsupervised word representations that only require large amounts of unlabeled text to learn (Mikolov et al., 2013b). Word2vec (Mikolov et al., 2013a) is a group of related models that are used to produce word embeddings. The vector representations of words learned by word2vec models have been shown to carry semantic meaning (Rong, 2014). Moreover, word2vec provides state-of-art word embeddings in various natural language processing (NLP) tasks (Goldberg and Levy, 2014). Therefore, we utilized word2vec to learn the word representations and compute the semantic similarities between comments and statements based on the learned word vectors.

Word2vec is a two-layer neural net that processes text. Its input is a text corpus and its output is a set of vectors: feature vectors for words in that corpus. Word vectors are positioned in the vector space such that words that share common contexts in the corpus are located in close proximity to one another in the space.

Word2vec can utilize either of two model architectures to produce a distributed representation of words: continuous bag-of-words (CBOW) or continuous skip-gram. In CBOW, the model predicts the current word from a window of surrounding context words. On the contrary, the skip-gram model uses the current word to predict the surrounding window of context words (Mikolov et al., 2013a). We use the latter method because it produces more accurate results on large datasets.

There are two tricks to optimize computational efficiency in Word2vec. One elegant approach to achieving this is an efficient way of computing softmax called hierarchical softmax and another approach is negative sampling (Mikolov et al., 2013a). The idea of

negative sampling is more straightforward than hierarchical softmax: in order to deal with the difficulty of having too many output vectors that need to be updated per iteration, we only update a sample of them. Therefore, we used negative sampling to optimize the training of the word vector model.

2.2. Random forests

One of the well-known methods in ensemble learning that generate many classifiers and aggregate their results is bagging of classification trees (Liaw et al., 2002). In bagging, each tree is independently constructed using a bootstrap sample of the dataset and a simple majority vote is taken for prediction in the end. Random forests, also known as random decision forests, are a popular ensemble method that can be used to build predictive models for both classification and regression problems (Breiman, 2001).

Random forests add an additional layer of randomness to bagging. Unlike traditional bagging, random forests change how the classification or regression trees are constructed. In standard trees, each node is split using the best split among all variables. In a random forest, each node is split using the best among a subset of predictors randomly chosen at that node. This somewhat counterintuitive strategy turns out to perform very well compared to many other classifiers, including discriminant analysis, support vector machines and neural networks, and is robust against overfitting (Liaw et al., 2002). Random Forests require a limited number of configuration parameters and produces a more stable model than basic decision trees (Genuer et al., 2010).

Specifically, the random forests algorithm first draws n bootstrap samples from the original data. Then, for each of the bootstrap samples, grow an unpruned classification tree, with the following modification: at each node, rather than choosing the best split among all predictors, randomly sample the predictors and choose the best split from among those variables. Finally, it predicts new data by aggregating the predictions of the n trees.

3. Approach

Our approach consists of three phases: We first prepared the dataset consisting of comment-statement pairs (Section 3.1). Then, we extracted important features from comments and statements (Section 3.2). Finally, we categorized the statements with machine learning and developed a strategy to determine the scopes of code comments automatically (Section 3.3).

3.1. Data preparation

This section describes the preparation of our dataset consisting of comment-statement pairs. There are three steps to prepare the dataset: the application of heuristic rules, comment selection and manual validation.

3.1.1. The application of heuristic rules

In this paper, we only focus on block and line comments within a method body, which are referred to as inline comments (comments for short in this paper) (Steidl et al., 2013). We first collected source code from open source projects and extracted comments from the source code. We then used heuristic rules to segment the code into comment and code pairs to facilitate manual validation. A set of heuristic rules are as follows:

(1) Adjoining comments are combined to be one comment because there is no code between them.
(2) The code snippet of a comment starts from the first line of the statement after the comment;
(3) The code snippet of a comment ends at the last line of the statement before another comment in the same block (not a sub-block), or the ending of the block, or the ending of a method.
(4) Comments nested in the sub-block of the statements are combined as one comment.

The heuristic rules are to segment the code roughly and it always segment the code to the end of the method, or the end of the block, or the begin of the next block/line comment in the same block. It is to maximum the scope of the comment. Algorithm 1 describes the details for source code segmentation by heuristic rules. We use these helper notations: a pair of comment and code is a quad $< cm, c, s, e >$ called *CC*, where *cm* and *c* represent a comment and its corresponding code snippet, and *s* and *e* are the number of start line and the end line of the code snippet respectively. Moreover, *startLine* is the number of the start line of a statement or a method. Similarly, *endLine* represents the number of the end line of a statement or a method.

Algorithm 1: The algorithm to segment the source code into comment and code pairs based on heuristic rules.

Input:
S: Source code; *C*: The set of extracted comments;
Output:
CCSet: The set of comment and code pairs;

```
1:  CCSet={};
2:  astTree = getASTTree(S);
3:  methodSet = getMethodSet(astTree);
4:  for each comment ∈ C do
5:      CC=(comment,null,0,0);
6:      for each method ∈ methodSet do
7:          if comment in method then
8:              codeSet={}; startLine=0; endLine=method.endLine;
9:              statementSet = getStatementSet(method);
10:             for each statement ∈ statementSet do
11:                 if statement.startLine>comment.startLine
                    && statement.endLine<endLine then
12:                     codeSet.add(statement);
13:                     if startLine == 0 then
14:                         startLine = statement.startLine;
15:                     end if
16:                 end if
17:                 if comment in statement && statement.endLine < end-
                    Line then
18:                     endLine = statement.endLine;
19:                 end if
20:             end for
21:             next_comment = getNextComment(comment,C);
22:             if next_comment.startLine - 1 < endLine then
23:                 endLine = next_comment.startLine - 1;
24:             end if
25:             CC.c=codeSet; CC.s=startLine; CC.e=endLine;
26:             goto 4;
27:         end if
28:     end for
29:     CCSet.add(CC);
30: end for
31: return CCSet
```

The algorithm inputs source code *S* and the set of extracted comments *C* from *S*, and outputs *CCSet*, which is the set of comment and code pairs. Firstly, we obtain the set of methods in *S* by building the Abstract Syntax Tree (AST) (Neamtiu et al., 2005) of *S* (lines 2, and 3). Secondly, for each comment in *C*, we find the method which it belongs to and set *e* of *CC* to the *endLine*

of the method as the default value in lines 7 and 8. Thirdly, the algorithm obtains the statement set of the method (lines 9) and searches the real *s* and *e* of *CC* (lines 10 to 25). Specifically, the *s* of *CC* is set to the *startLine* of the statement next to the comment and the statement is added to *c* of *CC* (lines 11 to 16). For the *e* of *CC*, if the statement contains the comment and *e* is greater than the *endLine* of the statement, we modify *e* to the *endLine* of the statement (lines 17, 19). If the *startLine* of the next comment in the same statement block is less than *e* of *CC*, then we modify it to the previous line of the *startLine* of the next comment (lines 21 to 24). lines 26 is to start a new code segmentation if it finishes searching the comment's code snippet in all the statements of the method. Finally, the algorithm of the heuristic rules returns *CCSet* (lines 31).

3.1.2. Comment selection

After applying the set of heuristic rules, we could get comment and code pairs from open source projects. But not all the comments have good quality for analysis. Moreover, not all the comments make descriptions for their following code snippets. Therefore, we filtered out four types of inappropriate comments as follows:

(1) **Code Comments**: Code Comments contain commented out code which is ignored by the compiler. Often code is temporarily commented out for debugging purposes or for potential later reuse.
(2) **Task Comments**: Task Comments are notes made by developers including TODO comments about functionalities to be implemented in next versions, FIXME comments about bugs that need to be fixed, NOTE comments about warnings and explanations, remarks about implementation hacks (Steidl et al., 2013).
(3) **IDE Comments**: IDE Comments are additional texts used to communicate with the IDE (Integrated Development Environment). They are often added automatically by the IDE or used by developers to change the default behavior of the IDE or compiler (Pascarella and Bacchelli, 2017).
(4) **Non-text Comments**: Non-text comments contain dates or websites that are irrelevant to the following code.

3.1.3. Manual validation

After the above steps, we obtained a number of comment and code pairs. However, the scopes of comments are inaccurate because of the simple heuristic rules that always segment extraneous statements into the scopes of comments. In other word, the resulting code snippet may be outside the scope of the comment. Therefore, based on the comment and code pairs after applying the heuristic rules, we manually validated the end line of the last statement within the scope of the comment to build the dataset of comment-statement pairs.

We used the webpage illustrated in Fig. 3 to validate the scope of a comment manually. The webpage comprises the following components:

(1) The information panel shows the names of the project and the class that the comment and code pair belongs to, and the ID of the comment as well.
(2) The code view panel is the main view of the webpage. It contains the comment and code pair and the context of the code in the method. Displaying the context of the segmented source code would be helpful for code reading and understanding. The comment for analysis is highlighted in blue and the code is formatted, which is to improve the quality of validation.
(3) The submission panel allows participants to submit the end line of the scope of the comment.

We invited 10 professional programmers having years of programming experience to determine the scopes of code comments through the webpage. Each pair of comment and code was validated by two participants. If the scopes of a comment determined by the two participants are inconsistent, we discarded it to purify the dataset.

3.2. Feature selection

This section introduces the selection of features to build the model based on machine learning. Whether the statement is in the scope of the comment can be influenced by many factors. We took a statement as an analysis unit and extracted features from the statement and its comment. The features are divided into three dimensions including code features, comment features and code-comment relationship features.

3.2.1. Code features

The first dimension is the code features. Code features describe the structural and the contextual information of a statement. Obviously, it is very important to understand the functionalities of the code snippets when documenting the code. Similarly, understanding the code and obtaining details of implementation contribute to determine whether the statement is in the scope of the comment or not.

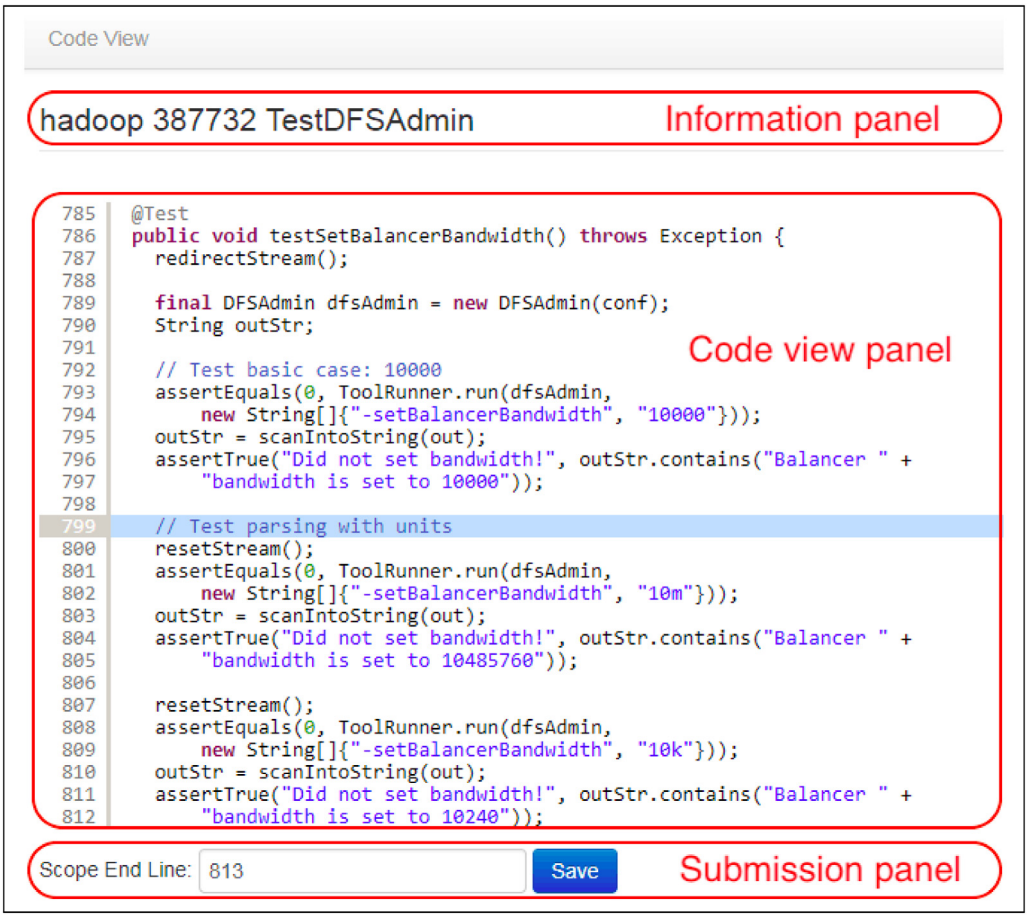
One of the structural information is the type of a statement. Different types of statements would have different behaviors and thus would have different impacts on the statement categorization. For example, a control statement such as a *For-statement* or *While-statement* appears after a comment is most likely to implement the function that mentioned in the comment. Therefore, we sampled 2000 comments from the dataset of comment-statement pairs and investigated the distribution of different types of statements within and outside the comment scopes. We obtained the abstract syntax tree (AST) (Neamtiu et al., 2005) of the Java source code utilizing Eclipse AST-Parser.[3] Based on the AST, we could extract different types of statements from the source code. As shown in Fig. 4, 81% of *For-statements* or *While-statements* are in the scopes of their corresponding comments, whereas 56% of *Return-statements* are out of their comment scopes. Thus, it indicates that different statement types effect differently on the statement categorization.

Another structural property of statements is the composition of a statement. In programming language, a statement can be composed by many sub-statements. In order to have a general understanding of the composition of a specific statement, we counted the number of sub-statements as well as the number of layers of statements nested in a statement. Intuitively, a complex statement with many sub-statements or many layers of nested statements is inclined to be explained by the comment writers for later code readers. Moreover, we also counted the number of lines of code in a statement.

For the contextual information, we analyzed method calls and variable usage in the context of a statement. If the statements use the same variable or invoke the same method, they tend to be an integral whole. Therefore, if a statement has the association of variables or method calls with statements that are in the scope of the comment, it is usually in the scope of the comment as well.

Finally, we checked whether there were blank lines before and after a statement. A professional programmer with proper programming style would use individual blank lines within the functions for separating the critical code sections to increase the functionality of the software (Vermeulen, 2000). Therefore, blank lines usually represent the ends of the scopes of comments.

[3] https://www.eclipse.org/jdt/.




```
@Test
public void testSetBalancerBandwidth() throws Exception {
  redirectStream();

  final DFSAdmin dfsAdmin = new DFSAdmin(conf);
  String outStr;

  // Test basic case: 10000
  assertEquals(0, ToolRunner.run(dfsAdmin,
      new String[]{"-setBalancerBandwidth", "10000"}));
  outStr = scanIntoString(out);
  assertTrue("Did not set bandwidth!", outStr.contains("Balancer " +
      "bandwidth is set to 10000"));

  // Test parsing with units
  resetStream();
  assertEquals(0, ToolRunner.run(dfsAdmin,
      new String[]{"-setBalancerBandwidth", "10m"}));
  outStr = scanIntoString(out);
  assertTrue("Did not set bandwidth!", outStr.contains("Balancer " +
      "bandwidth is set to 10485760"));

  resetStream();
  assertEquals(0, ToolRunner.run(dfsAdmin,
      new String[]{"-setBalancerBandwidth", "10k"}));
  outStr = scanIntoString(out);
  assertTrue("Did not set bandwidth!", outStr.contains("Balancer " +
      "bandwidth is set to 10240"));
```

Fig. 3. The webpage for validation.

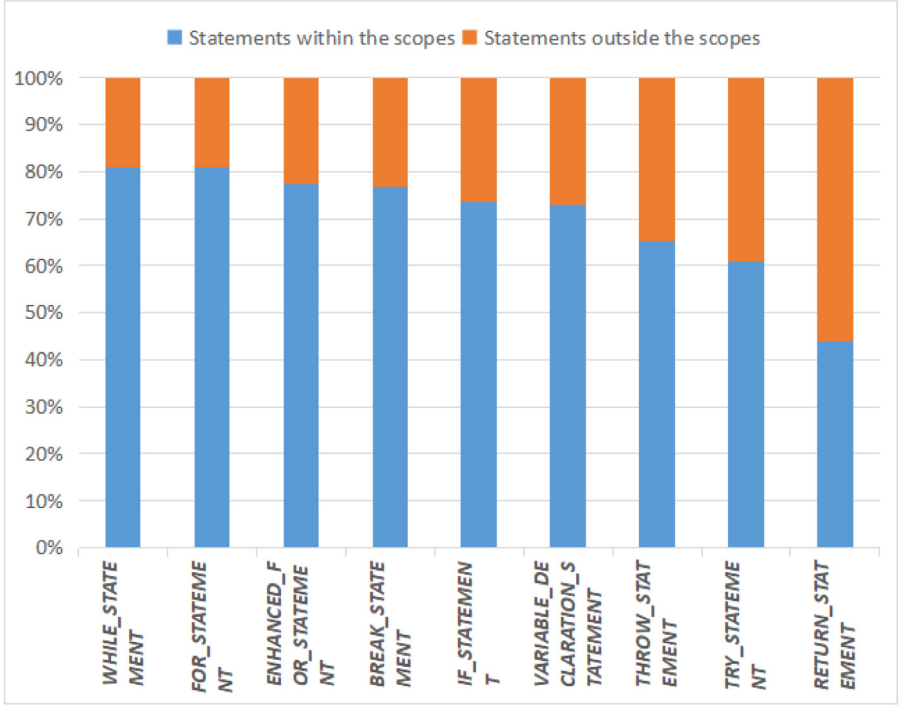


Fig. 4. Distribution of different types of statements within and outside the comment scopes.

Table 1
Code features.

No.	Features	Description
C_1	Statement Types	Nine common types of statements including *If-statements, While-statements, For-statements, EnhanceFor-statements, TryCatch-statements, Variable Declaration-statements, Return-statements, Break-statements, Throw-statements*
C_2	No. of sub-statements	The number of the sub-statements of the current statement
C_3	No. of layers of nested statements	The number of layers of statements nested in the current statement
C_4	No. of lines in the statement	The number of lines of code in the current statement
C_5	Same method calls	Does the current statement have the same method calls with the last statements and the following statements
C_6	Same variables	Does the current statement use the same variables with the last statements and the following statements
C_7	Preceding blank line	Is the preceding line of the current statement blank
C_8	Following blank line	Is the following line of the current statement blank

Table 2
Comment features.

No.	Features	Description
Cm_1	Length of the comment	The number of words in the comment
Cm_2	No. of verbs	The number of verbs in the comment
Cm_3	No. of nouns	The number of nouns in the comment
Cm_4	Following blank line	Is the following line of the comment blank

Eventually, we extracted eight types of code features, as listed in Table 1.

3.2.2. Comment features

The second dimension is the comment features. Comment features grab the information about comment words.

We first applied the following natural language processing techniques to preprocess comments:

(1) **Word Splitting**: Split code comments by white space. In some cases, comments may contain method names or identifier names. Thus, we split the single word based on camel case.
(2) **Stop word removal**: Remove code comments that contain any of the commonly used terms which are meaningless.
(3) **Word stemming**: Reduce inflected (or sometimes derived) words to their word stems, base or root forms with WordNet (Miller et al., 1990).

After that, we counted the number of words as well as the number of verbs and nouns in the comment. From the observation of commenting habits, we found that a comment followed by blank line is often the summary of a chunk of code. Therefore, we wondered whether the following line of a comment is blank or not and added it into comment features.

The comment features are listed in Table 2.

3.2.3. Relationship features

The third dimension is the relationship features. Relationship features described the correlation relationship between the comment and the statement.

The important feature we first considered was the textual similarity between the comment and the statement. There are divergent guidelines for how to write useful comments (McBurney and McMillan, 2016; Steidl et al., 2013). Comments are supposed to contain the intentions and goals of the implementation, and possible additional insights behind the implementation (Steidl et al., 2013). As a result, the description may mention the object or the functionality in the code snippet. Therefore, we focused on determining if the code comment contains text similarity terms that are specific to the statement.

In addition, a good comment written by developers should have a high semantic similarity to the source code. In general, comments have strong semantic associations with code snippets

write service filter link get ... map application target
time key manipulate interval new ... because zone function
⋮
frame item find index section ... layout position data

Fig. 5. A comment document. (Red words are randomly selected from code and green words are obtained from the corresponding comment). (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

(McBurney and McMillan, 2016). Hence, we calculated the semantic similarity between the comment and the statement. However, owing to the lexical gap between programming language and natural language, the semantic similarity cannot be accurately measured by word embeddings directly. To overcome this issue, we developed a skip-gram model based on the method proposed by Ye et al. (2016), which bridges the lexical gap by projecting natural language statements and code snippets as meaning vectors in a shared representation space.

To build the corpus for the training of word vector representations, we first crawled 1000 popular Java projects from Github using its search API.[4] Then we segmented the source code by Algorithm 1 and preprocessed the source code and comments from the resulting comment and code pairs using the natural language processing techniques, which has been described in the previous section. Then, according to the linguistic hypothesis (Harris, 1954), words that appear in the same contexts tend to have similar semantic meanings. Therefore, for each word in the comment, we randomly selected two words in the statement and added them to the comment as a comment document. Each word of comment in comment document is surrounded by two random words from the statement as shown in Fig. 5. Next, for a statement, we randomly selected two words from the associated comment and added them to the statement to form the document of statements. Finally, these two documents were merged and served as the corpus of our skip-gram model (Mikolov et al., 2013b).

Based on the corpus, we trained our skip-gram model and obtained the vector representation of each word. The continuous skip-gram is effective at predicting the surrounding words in a context window of $2k+1$ words, where $k=2$ and the window size is 5 in our model. The objective function of the skip-gram model aims at maximizing the sum of the log probabilities of the surrounding context words conditioning on the central word (Mikolov et al., 2013b):

$$\sum_{i=1}^{n} \sum_{-k\le j\le k, j\neq 0} \log p\left(w_{i+j}|w_i\right) \tag{1}$$

where w_i and w_{i+j} denote the central word and the surrounding context words within a context window of length $2k+1$ respectively and n denotes the length of the word sequence. The term

[4] https://developer.github.com/v3/search/.

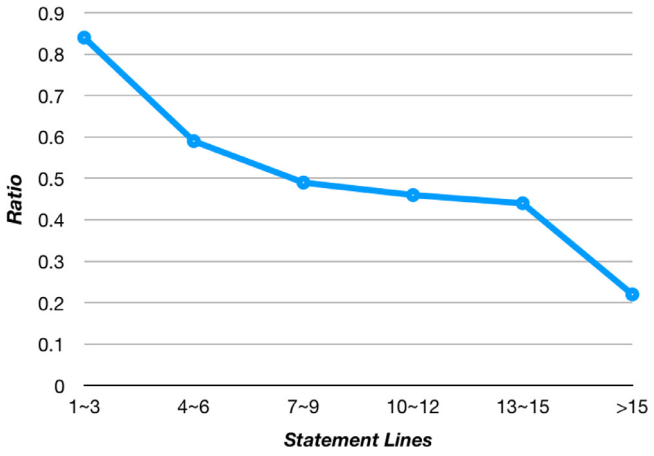


Fig. 6. Ratios of statements within the scopes of comments under different distances between comments and statements.

$\log p(w_{i+j}|w_i)$ is the conditional probability, which is defined using the *Softmax* function:

$$\log p(w_{i+j}|w_i) = \frac{\exp\left(v'^{T}_{w_{i+j}} v_{w_i}\right)}{\sum_{w\in W}\exp\left(v'^{T}_{w} v_{w_i}\right)} \tag{2}$$

where v_w represents the input vector and v'_w represents the output vectors of word w in the underlying neural model. W denotes the vocabulary of all the words. Intuitively, $p(w_{i+j}|w_i)$ estimates the normalized probability of word w_{i+j} appearing in the context of central word w_i over all the words in the vocabulary. We employed a negative sampling method (Mikolov et al., 2013b) to optimize computational efficiency when computing this probability.

To compute the semantic similarities between comments and statements, we defined three types of similarity measures:

(1) **Word to word**: given two words w_1 and w_2, we defined their semantic similarity as the cosine similarity between their learned word embeddings:

$$sim(w_1, w_2) = \cos(\mathbf{w}_1, \mathbf{w}_2) = \frac{\mathbf{w}_1^T \mathbf{w}_2}{||\mathbf{w}_1||||\mathbf{w}_2||} \tag{3}$$

(2) **Word to sentence**: given a word w and a sentence S, the similarity between them is computed as the maximum similarity between w and any word w' in S:

$$sim(w, S) = \max_{w'\in S} sim\left(w, w'\right) \tag{4}$$

(3) **Sentence to sentence**: between a comment C and a statement S, we define the semantic similarity as follows:

$$sim(C, S) = \frac{sim(C \to S) + sim(S \to C)}{2} \tag{5}$$

where

$$sim(C \to S) = \frac{\sum_{w\in C} sim(w, S)}{n} \tag{6}$$

where n denotes the number of words in C.

Besides, we investigated the question that whether the statement is in the scope of the comment is related to the distance between the statement and the comment. As shown in Fig. 6, the probability of a statement in the scope of the comment decreases as the distance increases. It indicates that the farther away the statement is, the less likely the statement is to be within the scope of the comment. Therefore, we added two distance measures between the comment and the statements by counting the number of lines of code and the number of statements between them.

The relationship features are summarized at Table 3.

3.3. Detection of comment scopes

In order to classify statements into two categories: within and outside the scopes of the comments, we utilized a supervised machine learning algorithm called random forests. The random forests algorithm has better prediction accuracy than basic decision trees and other advanced machine learning algorithms. Moreover, the random forests algorithm requires a limited number of configuration parameters and produces a robust and stable model. Finally, the fandom forests algorithm deals well with correlated features while maintaining a high accuracy in its prediction, which can simplify our feature selection.

After statement categorization, we applied a strategy to finally determine the scope of a comment. For statements in a comment-statement pair, we regarded the first out-of-scope statement as the demarcation point of the scope of the comment. In other words, the end line of the statement before the first out-of-scope statement is the end of the scope of the comment. There is a more explicit example in Fig. 7. As we can see, line 5 is classified as the first out-of-scope statement, so the scope of the comment is line 2 to line 3. It means that we have to classify all the statements in the scope of the comment correctly.

4. Evaluation

We evaluated our technique as a comment scope detection method in general and its application to two software engineering tasks in particular. There are two reasons for choosing these two tasks. First, both two tasks utilized the information of the scopes of block/line comments. When mapping the code to the corresponding block/line comments, they both suffered from the inaccuracy of the heuristic rules on comment scope detection. Second, their source code and data sets are available, thus we can reimplement and try to improve them. For that purpose, we performed three empirical studies. In the first study, we inspected the effectiveness and robustness of our method by computing the *accuracy* of the method, and analyzing the effect of the feature selection and the sizes of comment scopes as well. Finally, we compared the same measures of our method on brand new projects. In the second study, we applied our method to comment-code consistency detection, and compared its performance against the original method proposed by Liu et al. (2018). In the third study, we applied our method to CloCom proposed by Wong et al. (2015) to find out whether our method contributes to automatic comment generation by mining software repositories.

Table 3
Relationship features.

No.	*Features*	*Description*
R_1	Textual similarity	The number of common key words in the comment and the statement
R_2	Semantic similarity	The cosine distance between the comment and the statement
R_3	Line distance	The number of lines of code between the comment and the current statement
R_4	Statement distance	Number of statements between the comment and the current statement

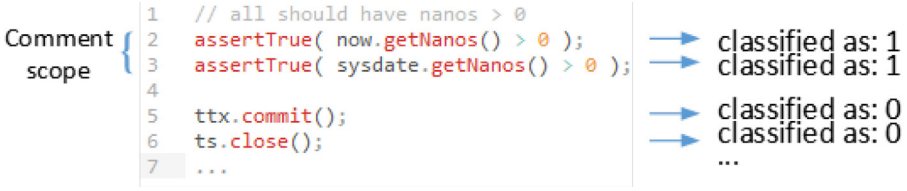


Fig. 7. An example of our strategy to determine the comment scope.

Table 4
Details of the four open source projects.

Projects	*LOC*	*No. of files*	*No. of block/line comments*
Hadoop	2,275,828	9578	46,567
Hibernate	989,658	9152	11,739
JDK	2,322,944	7691	35,403
JEdit	184,325	601	5987
TOTAL	5,772,755	27,022	99,696

Accordingly, we formulated the following six research questions:

RQ1: How effective is our method in detecting the scopes of comments?

RQ2: What is the effect of the three dimensions of features?

RQ3: Do the sizes of scopes of comments affect the accuracy of our method?

RQ4: How does our method perform on new projects without prior knowledge?

RQ5: Can our method improve the performance of outdated comment detection during code changes?

RQ6: Does our method contribute to automatic comment generation when mining existing source code from open source projects?

4.1. Study I: effectiveness and robustness of our comment scope detection method

This section presents the main study, which addresses the effectiveness and robustness of the our comment scope detection method. We studied the *accuracy* of our method in different learning scenarios. This study aims to answer the first four research questions (RQ1-RQ4).

4.1.1. Data preparation

In order to build the dataset of comment-statement pairs, we extracted source code from fined-grained projects in Java. Specifically, we used the following four open source projects: Hadoop[5], Hibernate, JDK[6], and JEdit. They are considered to be well documented, have high-quality comments, and be widely used in related studies (Liu et al., 2018; Huang et al., 2017b). Table 4 details the selected systems.

The four projects emerge from the context of different software ecosystems to mitigate the external validity. Hadoop is a framework that allows for the distributed processing of large data sets across clusters of computers using simple programming models. Hibernate (Hibernate ORM) is an object-relational mapping tool which provides a framework for mapping an object-oriented domain model to a relational database. JDK (Java Development Kit) is a development environment for building applications and components in Java. JEdit is a mature programmer's text editor with hundreds of person-years of development.

[5] http://hadoop.apache.org/.

[6] http://www.oracle.com/technetwork/java/javase/overview/index.html.

Then, we prepared the dataset as described in Section 3.1. We segmented the source code into comment and code pairs by the heuristic rules and filtered out inappropriate comments. Because of the inaccuracy of the heuristic rules, we used the webpage illustrated in Fig. 3 to validate the scope of a comment manually. Eventually, we got 1000 comment-statement pairs from each project and there are 4000 pairs in total in our dataset. Specially, the dataset contains 226 comments whose scopes have been manually marked by the developers: 151 comments from jEdit and 75 comments from hibernate.

4.1.2. Effectiveness metrics

To evaluate the effectiveness of our automated technique on detecting the scopes of comments, we first used three well-known Information Retrieval (IR) metrics for the quality of the statement classification results (Schütze, 2008), named *precision, recall* and *f-score*. And then we calculated the *accuracy* of our method in detecting the scopes of comments after applying the scope strategy.

(1) *precision* and *recall*. We computed the *precision* and *recall* of our comment scope detection method as following:

$$precision = \frac{|TP|}{|TP + FP|} \times 100\% \tag{7}$$

$$recall = \frac{|TP|}{|TP + FN|} \times 100\% \tag{8}$$

Where $|TP|$ is the number of true positives (positive instances correctly categorized), $|FP|$ is the number of false positives (positive instances incorrectly categorized), $|TN|$ is the number of true negatives (negative instances correctly categorized), and $|FN|$ is the number of false negatives (negative instances incorrectly categorized). Therefore, the precision is the percentage of positive instances identified by our random forests classifier that are actually positive instances. The recall is the percentage of true positive instances that are successfully retrieved by our classifier.

(2) *f-score*. It computes the harmonic mean of *precision* and *recall* to weight them evenly, where an *f-score* reaches its best value at 1 (perfect precision and recall) and worst at 0. Mathematically,

$$f\text{-}score = \frac{2 * precision * recall}{precision + recall} \times 100\% \tag{9}$$

(3) *accuracy*. After statement categorization and applying the scope strategy, we calculated the *accuracy* of our method in detecting the scopes of comments as following:

$$accuracy = \frac{|Correct\ Comments|}{|Total\ Comments|} \times 100\% \tag{10}$$

Where $|Correct\ Comments|$ is the number of comments that our method detects their scopes correctly and $|Total\ Comments|$ is the total number of comments being detected.

4.1.3. Results

RQ1: How effective is our method in detecting the scopes of comments?

Table 5
Dataset of comment-statement Pairs.

Projects	*No. of pairs of comment-statement*	*No. of statements*		
		Within the scope	*Outside the scope*	*Total*
Hadoop	1000	2825	1211	4036
Hibernate	1000	3473	1299	4772
JDK	1000	2066	677	2743
JEdit	1000	2500	2214	4714
TOTAL	4000	10864	5401	16265

Table 6
Evaluation of different algorithms.

Algorithms	*Within the scope*			*Outside the scope*			*Accuracy*
	Precision	*Recall*	*F-score*	*Precision*	*Recall*	*F-score*	
J48	86.57%	87.78%	87.12%	71.92%	70.37%	70.97%	76.25%
SMO	83.53%	90.46%	86.83%	73.47%	60.13%	65.99%	79.95%
Naive Bayes	78.24%	89.28%	83.32%	62.48%	40.51%	47.84%	64.93%
Logit Boost	84.52%	90.56%	87.40%	74.20%	62.51%	67.74%	76.85%
AdaboostM1	92.34%	84.55%	88.19%	71.05%	84.47%	76.96%	81.35%
Random Forests	90.49%	88.20%	89.29%	74.97%	79.51%	77.01%	81.45%

We would like to investigate how accurate our method can achieve in detecting the scopes of comments. First, we trained a classifier to categorize the statements into two types: statements within the scope of the comment and statements outside the scope of the comment. In other words, based on the comment-statement pairs, we viewed statements before the end line of the last statement within the scope of the corresponding comment as positive instances, otherwise as negative ones. As listed in Table 5, we obtained 10,864 positive instances and 5401 negative instances from 4000 comment-statement pairs.

We used a similar 10-fold cross-validation approach to validate our classifier. According to the 10-fold cross evaluation mechanism, the dataset was randomly partitioned into 10 subgroups and each subgroup contained 400 comment-statement pairs. Of the 10 subgroups, a single subgroup was retained as the validation data for testing the model, and the remaining 9 subgroups were used as training data. Then based on the ten pairs of training and testing data, we trained and evaluated 10 models. The results from the ten models were averaged to produce a single estimation.

Then we compared different machine learning approaches on the dataset of the four projects and obtained quantitative measurements of how well they performed on the detection of comment scopes. The main purpose was to choose a best classifier from numerous machine learning techniques, including J48, SMO, Naive Bayes, Logit Boost, AdaboostM1, and random forests. In Table 6, the result showed that the most effective machine learning algorithm for comment scope detection is random forests. The random forest algorithm, which has the highest *f-scores* of both positive and negative samples, can automatically identify the most informative features from the numerous of features. It is important to screen out the most useful features to train a model with stronger classification capacity.

Finally, we utilized random forests to build our model. When training, we set the number of decision trees in the random forests algorithm as 150 and the number of features to use in random selection as 13 to obtain the best model. As shown in Table 6, the *precision* and the *recall* of the positive instances reached 90.49% and 88.20% respectively, and the *f-score* is as high as 89.29%. Our method exhibited relatively poor performance on negative instances, and the values of *precision* and *recall* were 74.97% and 79.51% respectively, and the *f-score* is 77.01%. The poor performance on negative instances may be caused by the unbalanced positive and negative instances in the dataset that the number of positive instances were almost two times the number of negative instances.

Eventually, we applied our scope strategy described in Section 3.3 to determine the scope of each comment based on the result of statement categorization. As shown in Table 7, the total *accuracy* of our method to detect the scope of comment was 81.45%. Moreover, compared to the heuristic rules, the *accuracy* of our method had an improvement of approximately 14.3%, increasing from 67.10% to 81.45%. It indicates that our method is effective in detecting the scopes of comments.

Answer to RQ1: *Our method, which utilizes machine learning, is effective in detecting the scopes of comments with an accuracy of 81.45%.*

RQ2: What is the effect of the three dimensions of features?

To answer RQ2, we evaluated the effect of different dimensions of features (i.e., code features, comment features and relationship features) and different feature combinations on detecting the scopes of comments. The result was obtained by random forests with 10-fold cross validation as well, as presented in Table 8.

Generally, every dimension of features is important. And the most valuable features are the code features. From the table we can see that when only applying code features to train learning model, the *precision, recall* and *f-score* of negative instances are 72.14%, 77.29% and 74.46%, and our method achieved an *accuracy* of 78.55% in detecting the comment scopes. Moreover, when features from different dimensions are combined in pairs, they perform better than individual features. For example, when the code features are combined with either the comment features or the relational features, the performance of our model is improved. Specifically, when combining code features with relationship features, the values of the *precision, recall, f-score* of negative instances and *accuracy* in comment scope detection exhibited obvious improvement, increasing to 74.46%, 77.63%, 75.85% and 80.32%, respectively. Lastly, when adding the comment features into the learning model, the values of *precision* and *recall* of negative instances have increased slightly to 74.97% and 79.51%, and our method achieved its highest *f-score* (77.01%) of negative instances. Moreover, the detection *accuracy* has increased to 81.45%. Although there were no significant differences among the *precision, recall* and *f-score* of positive instances, the result confirmed that the three dimensions of features we selected are useful in characterizing the

Table 7
Comment scope detection result.

Approach	*No. of correctly detected comments*	*No. of detected comments*	*Accuracy*
Heuristic rules	2684	4000	67.10%
Ours	3258	4000	81.45%

Table 8
Effect of three dimensions of features.

Features	*Within the scope*			*Outside the scope*			*Accuracy*
	Precision	*Recall*	*F-score*	*Precision*	*Recall*	*F-score*	
Code	89.32%	86.63%	87.89%	72.14%	77.29%	74.46%	78.55%
Comment	74.45%	35.30%	47.59%	36.33%	74.57%	48.72%	63.35%
Relationship	87.09%	85.92%	86.47%	68.68%	70.33%	69.38%	70.63%
Code + Comment	89.44%	87.21%	88.25%	73.05%	77.66%	75.10%	78.95%
Code + Relationship	89.83%	88.21%	88.96%	74.46%	77.63%	75.85%	80.32%
Comment + Relationship	75.74%	85.66%	80.37%	60.19%	44.30%	50.90%	64.30%
Code + Comment + Relationship	90.49%	88.20%	89.29%	74.97%	79.51%	77.01%	81.45%

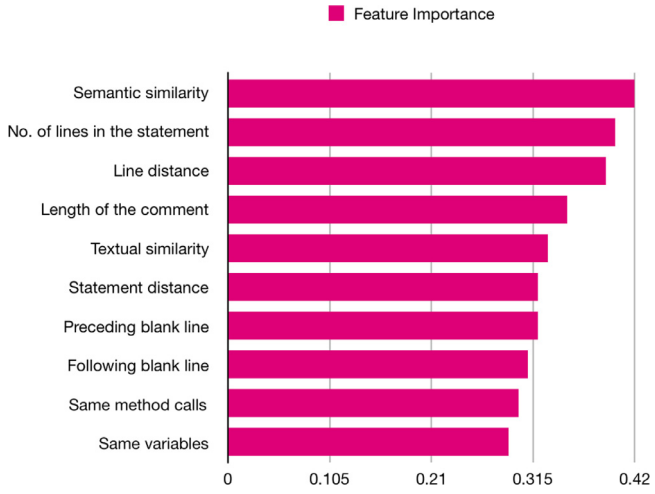


Fig. 8. The top 10 most important features.

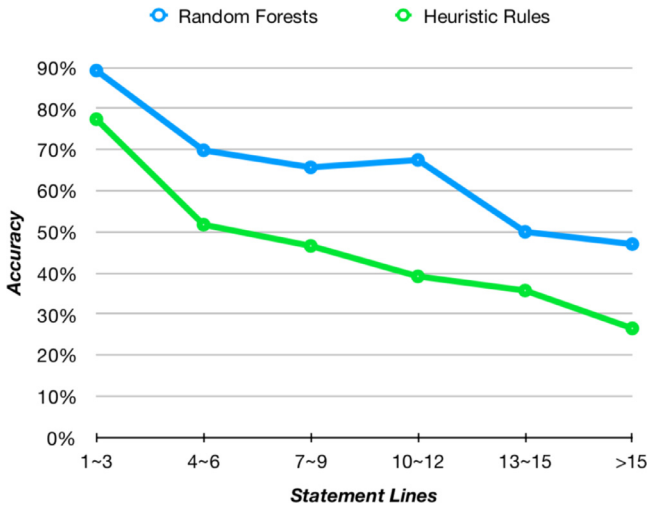


Fig. 9. Comparison of accuracy between the two methods under different comment scope sizes.

logical strength of comments and statements from three different perspectives. It indicated that the three dimensions have positive impacts on detecting the comment scopes.

In addition, we utilized random forests algorithm to calculate the importance of selected features and rank them. Fig. 8 illustrated the top 10 features. As shown in the figure, the most importance feature is the semantic similarity between the comment and the statement. As we can expect, when a statement is semantically similar to a comment, it is likely to be within the scope of the comment.

Answer to RQ2: *The three dimensions of features we selected are all useful in characterizing the logical strength of comments and statements from three different perspectives, and play a positive role in comment scope detection.*

RQ3: Do the sizes of scopes of comments affect the accuracy of our method?

We would like to know which sizes of scopes of comments that can make our method yield optimum results. Therefore, we compared the performance of our methods on different number of statements within the scopes of comments. Fig. 9 illustrated the result.

As we can observe, when the size of the scopes of the comment is 1 to 3, our method achieved the highest *accuracy* of 89.27% on detecting the scopes of comments. Namely, when the scopes of comments contain 1 to 3 statements, our method achieved the best results in detecting the scopes of comments. Compared to the method that only applied heuristic rules on comment scope detection, the performance of our method exceeded 11.85% when the sizes of the comment scopes are 1 to 3 statements. When the sizes of the scopes of comments increase, the *accuracy* of our method declines because the ratio of comments with large scopes is relatively small as illustrated in Fig. 6. Another reason for lower accuracies in large comment scopes is that comments with large scopes tend to be abstract and do not have much semantic information. Moreover, as the sizes of the scopes of comments increase, the curve of our method remains above the curve of the heuristic rules. It indicated that the *accuracy* of our method can be affected by the sizes of the scopes, and our method performs stronger than the heuristic rules when the scopes of comments cover a large number of statements.

Answer to RQ3: *The sizes of scopes of comments would affect the accuracy of our comment scope detection method. But compared to heuristic rules, our method performs better when the scopes of comments cover a large number of statements.*

RQ4: How does our method perform on new projects without prior knowledge?

The above experiments provided promising results under the setting with prior knowledge, strongly indicating that our method would likely work well in the scenario of developing some new components in old projects that used to build the model. However, many real-world projects are small or new, and training data for

Table 9
Evaluation result of our method on new projects.

Projects	*Within the scope*			*Outside the scope*			*Accuracy*
	Precision	*Recall*	*F-score*	*Precision*	*Recall*	*F-score*	
Hadoop	88.52%	92.78%	90.60%	81.02%	71.92%	76.20%	79.80%
Hibernate	89.30%	85.55%	87.38%	65.26%	72.59%	68.73%	79.50%
JDK	90.87%	90.13%	90.50%	70.61%	72.38%	71.48%	82.30%
JEdit	94.31%	80.92%	87.10%	81.43%	94.49%	87.48%	81.30%
Average	90.75%	87.35%	88.90%	74.58%	77.85%	75.97%	80.73%

model construction is limited. Even if the amount of data for the new project is large, building training sets from the new project can be time-consuming. In these cases, it is valuable to explore whether our method can perform well on a brand new project.

Therefore, in order to create a new project scenario on the same dataset in Table 5, we used the 3000 comment-statement pairs of the three projects as the training set and the 1000 comment-statement pairs of the remaining project as the test set. The evaluation results for each project are provided in Table 9.

In contrast to above experiments, the learning scenario without prior knowledge is significantly more challenging. However, our method performed effectively on brand new projects. Although the *accuracy* of Hibernate is relatively lower, our method achieved an average *accuracy* of 80.73%. Specially, our method on JDK and JEdit outperformed the average accuracy, and up to an *accuracy* of 82.30%. The result indicated that the our method did not performed differently when applying to new projects, although different projects may follow different commenting practices and some project-specific knowledge is challenging to adapt to other projects. At the same time, the result demonstrated the feasibility and effectiveness of our comment scope detection method.

Answer to RQ4: *The performance of our method on new projects is considerably effective, which indicates that our method is a general approach in detecting the scopes of comments.*

4.2. Study II: applications to software repository analysis for outdated comment detection

This section presents our second study, which addresses the application of our method in software repository analysis for outdated comment detection. This study concerns the performance benefits of our comment scope detection method to the approach on outdated comment detection proposed by Liu et al. (2018) and seeks to answer the fifth research question (RQ5). That is, we expect that our method improves the performance on the detection of outdated comments.

4.2.1. Methodology

To evaluate the effectiveness of our general method to detect the scopes of comments, we applied our method to the previous approach on out-of-date comment detection on software repositories. We then compared the performance between the baseline method and the improved method using our method in comment scope detection step. The baseline method, proposed by Liu et al. (2018), was a machine learning based method for detecting the comments that should be changed during code changes. Their method focused on the automatic detection of the outdated block/line comments between two code versions.

As illustrated at Fig. 10, there were five main steps in their approach as following:

(1) **Comment Scope Detection**: From software repositories like Github[7], they first obtained two versions of source code in a single commit, and then used a set of heuristic rules mentioned in Section 3.1.1 to deduce the scopes of comments in this two versions.
(2) **Dataset Construction**: After determined the scopes of the comments, they built the dataset which consists of two versions of comment and code pairs, and automatic labeled the dataset by comment states (changed or remained the same) between two versions.
(3) **Software Change Extraction**: Based on the dataset, they extracted code changes such as statement insertion between two versions of source code within the scopes of the comments by a tool called ChangeDistiller (Gall et al., 2009).
(4) **Feature Extraction**: In this step, they extracted 64 discriminative features from code changes, comments, and the relationship between the comments and code for machine learning.
(5) **Random Forest Classifier**: Finally, they trained a random forest classifier to detect outdated comments during code changes.

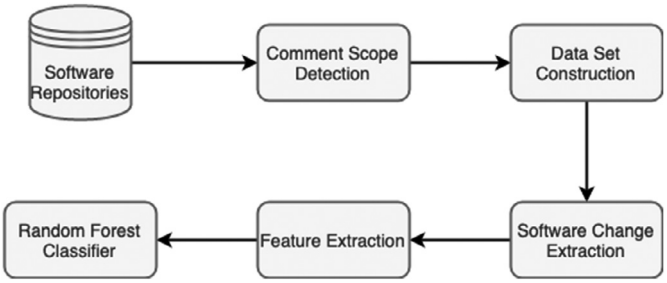


Fig. 10. The approach of the detection of comment-code inconsistencies.

However, in the comment scope detection step, they mentioned that using heuristic rules to determine the scope of a comment may not be precise, because a comment may cover its following statements that are outside the scopes (Liu et al., 2018). In fact, as validated in Study I, the *accuracy* of heuristic rules on comment scope detection is too low to be satisfied. As a result, their dataset is not clear as there are many irrelevant statements outside the scopes of the comments, which would have a negative effect on the results of the detection of comment-code inconsistencies.

Therefore, to detect outdated comments at a fine-grained level, we substituted heuristic rules with our method in the comment scope detection step, and utilized machine learning technique to determine whether the comments should be changed. Our approach is the same in Fig. 10 except for the comment scope detection methods. To evaluate the effectiveness of our method, we compared the *precision, recall* and *f-score* against Liu's approach.

4.2.2. Data preparation

We used the same dataset collected by Liu et al. (2018). In their dataset, 33,050 software changes were collected from 5 open

[7] https://github.com/.

Commit 22361: Old Version

```
1  //{{{ ensure we don't have
       empty space at the
       bottom or top, etc
2  int max = displayManager.
       getScrollLineCount() -
       visibleLines + (
       lastLinePartial ? 1 :
       0);
3  if(firstLine > max)
4    firstLine = max;
5  if(firstLine < 0)
6    firstLine = 0;
7  //}}}
8
9
10
11 if(Debug.SCROLL_DEBUG)
12 {
13   Log.log(Log.DEBUG,this,"
         setFirstLine() from "
14   + getFirstLine() + " to "
           + firstLine);
15 }
16 ...
```

Commit 22361: New Version

```
1  //{{{ ensure we don't have
       empty space at the
       bottom or top, etc
2    int max = displayManager.
         getScrollLineCount()
         - visibleLines + (
         lastLinePartial ? 1 :
          0);
3    if(firstLine > max)
4      firstLine = max;
5    if(firstLine < 0)
6      firstLine = 0;
7  //}}}
8
9  int oldFirstLine =
       getFirstLine();
11 if(Debug.SCROLL_DEBUG)
12 {
13   Log.log(Log.DEBUG,this,"
         setFirstLine() from "
14   + oldFirstLine + " to " +
           firstLine);
15 }
16 ...
```

Fig. 11. A negative instance from Liu's dataset: two versions of codes of JEdit(Code changes are highlighted).

source projects including JEdit, OpenNMS[8], JAMWiki[9], EJBCA[10] and JHotDraw.[11] They are widely used in source code analysis research papers (Dit et al., 2013; Huang et al., 2017a) and have different application domain types, including text editor, managing system, collaborative software, Wiki engine, and two-dimensional graphical framework, which would mitigate the external validity. Among these software changes, 28,730 comments stayed the same, and 4320 comments were changed. Thus, these 4320 comments were treated as outdated comments.

However, their dataset contains a number of noisy data owing to the inaccurate heuristic rules in comment scope detection. For example, Fig. 11 shows a negative sample from their dataset. The result of our comment scope detection method showed that the scope of the comment *"ensure we don't have empty space at the bottom or top, etc"* covers line 2 to line 7, which is consistent to the scope markers made by the authors of JEdit. Intuitively, the code changes between the two versions of source code, which is highlighted in the figure, are outside the scope of the comment and have no effects on the changes of comment states. In other words, we discarded it because there is no code change within the scope of the comment after applying our comment scope detection method.

Moreover, as illustrated in Fig. 12, they combined the inserted comment *"Get a decent source of random data"* to the comment *"get home interfaces to other session beans used"* when inserting a code snippet in the new code version, and they treated it as a positive instance. However, based on the atomic principle, it is wrong as the inserted comment and code snippet are independent to the code and the comment in the older version. The code within the scope of the comment does not change as well. Thus, after applying our method, we filtered out 11,380 noisy data. Among them, 1513 outdated comments and 9867 up-to-date comments were discarded. As a result, there are 2807 positive instances and 18,863 negative instances in our dataset as shown in Table 10.

4.2.3. Results and analysis

RQ5: Can our method improve the performance of outdated comment detection during code changes?

To answer RQ5, we performed an experiment on the dataset and compared performance of our method against Liu's. Specifically, after applying our comment scope detection method, we filtered out some noisy data, then extracted software changes from the new constructed dataset. Next, we extracted as many as 64 features from these software changes after applying our comment scope detection method, and finally trained a random forest classifier to detect outdated comments. Because there is a large disparity in the proportion of positive and negative samples in the dataset, we also introduced the cost matrix (He and Garcia, 2009) in the random forest classifier and used the best parameter setting mentioned in their paper. The form of the cost matrix is represented as follows:

$$\begin{bmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{bmatrix}$$

where C_{ij} indicates the cost of the misclassification error when the classifier misclassifies i as j. In general, $C_{00} = C_{11} = 0$. After parameter tuning, we set C_{10} as 6 and C_{10} as 1. Finally, we used 10-fold cross-validation to evaluated our models and compared the metrics of *precision, recall* and *f-score* between the two approaches.

Table 11 shows the comparison results between our method and Liu's method. We compared the results of improved method

Table 10
Datasets before and after application of our method in the detection of outdated comments.

Datasets	*Before application*	*After application*	*Difference*
No. of Out-of-date Comments	4320	2807	1,513
No. of Up-to-date Comments	28,730	18,863	9,867
TOTAL	33,050	21,670	11,380

[8] http://www.opennms.org.
[9] http://www.jamwiki.org.
[10] http://www.ejbca.org.
[11] http://www.jhotdraw.org.

Commit 1077: Old Version

```
// get home interfaces to other session beans used
storeHome = (ICertificateStoreSessionLocalHome) lookup("java:
    comp/env/ejb/CertificateStoreSessionLocal");
authHome = (IAuthenticationSessionLocalHome) lookup("java:comp
    /env/ejbAuthenticationSessionLocal");

ILogSessionHome logsessionhome = (ILogSessionHome) lookup("
    java:comp/env/ejb/LogSession",ILogSessionHome.class);
logsession = logsessionhome.create();
```

Commit 1077: New Version

```
// get home interfaces to other session beans used
storeHome = (ICertificateStoreSessionLocalHome) lookup("java:
    comp/env/ejb/CertificateStoreSessionLocal");
authHome = (IAuthenticationSessionLocalHome) lookup("java:comp
    /env/ejb/AuthenticationSessionLocal");

ILogSessionHome logsessionhome = (ILogSessionHome) lookup("
    java:comp/env/ejb/LogSession",ILogSessionHome.class);
logsession = logsessionhome.create();

// Get a decent source of random data
String  randomAlgorithm = (String) lookup("java:comp/env/
    randomAlgorithm");
randomSource = SecureRandom.getInstance(randomAlgorithm);
```

Fig. 12. A positive instance from Liu's dataset: two versions of codes of EJBCA(Code changes are highlighted).

Table 11
Approach comparison in the detection of outdated comments.

Approach	*Out-of-date comment*			*Up-to-date comments*		
	Precision	*Recall*	*F-Score*	*Precision*	*Recall*	*F-Score*
Baseline	77.2%	74.6%	75.9%	96.4%	97.2%	96.8%
Ours	83.0%	78.8%	80.8%	96.9%	97.6%	97.2%

to their results reported in the paper. According to the table, our method outperformed the previous approach for all the metrics. Especially for the positive samples, the *precision* increased from 77.2% to 83.0%, and the *recall* increased from 74.6% to 78.8% as well. Moreover, our method obtained a *f-score* (80.8%), a noticeably higher score than Liu's 75.9% for the positive samples. This considerable result indicates that most part of outdated comments could be correctly categorized and retrieved when applying our method.

In conclusion, our comment scope detection method is more effective than the heuristic rules, which is the reason that our method achieved a great improvement on outdated comment detection. It can be observed that our method is more preciser than the heuristic rules in detecting the scopes of comments, thus most of the potential noisy data was removed in the dataset. That is, our method discarded some irrelevant code changes which is outside the scope of the comment. In this way, our method help the random forest model learning common outdated comment knowledge more effectively.

Answer to RQ5: *As a general method, our method can greatly improve the performance of out-of-date comment detection during code changes. It indicates that our method is conducive to comment-code relationship analysis.*

4.3. Study III: applications to software repository mining for comment generation

This section presents our third study, which addresses the application of mining software repositories for automatic comment generation. This study concerns the efficiency benefits of our comment scope detection method to Clocom, a tool proposed by Wong et al. (2015) to generate comments by mining software repositories, and seeks to answer the last research question (RQ6). That is, we expect that our method help to raise the ratio of comments that are manually evaluated as good during the comment generation process.

4.3.1. Methodology

In order to further verify the effectiveness of our general method to detect the scopes of comments, we applied our method to a comment generation tool called CloCom (Wong et al., 2015). We then compared the results between CloCom and the improved CloCom with our method (referred to as Improved-CloCom in the following). CloCom, proposed by Wong et al. (2015), is a automatic comment generation tool by analyzing existing software repositories. It applies code clone detection techniques to discover similar code segments between the input target project and the software repositories. After that, it reuses the comments from these cloned code segments in the software repositories to describe the similar code segments from the target project. As illustrated at Fig. 13, CloCom takes two inputs: (1) software projects for comment extraction, e.g., open source projects from GitHub, and (2) a target software project to be commented. The output of CloCom is a list of automatically generated comments for the cloned code snippets from target project. There were five main steps in CloCom as following:

(1) **Code Clone Detection**: It detects code clones between the database containing raw software projects and the target projects' source code to discover similar code segments.
(2) **Code Clone Pruning**: The code clone detection step simply reports a match if two code segments are syntactically similar. Therefore, CloCom has to prunes out code clones that do not have semantic similarity. Specifically, it filters out code clones that are syntactically similar, but not semantically similar using basic heuristics.

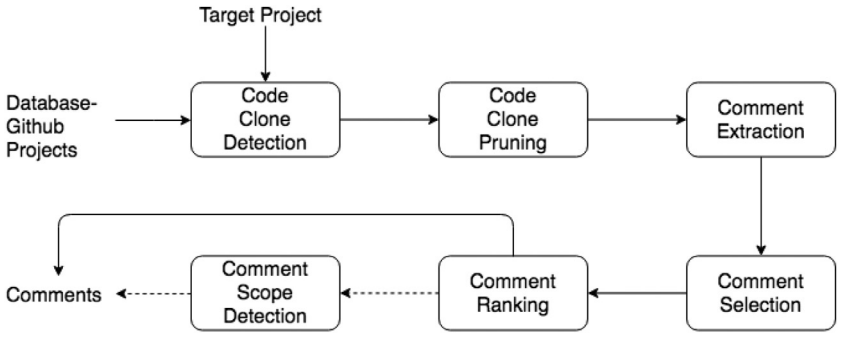


Fig. 13. Overviews of CloCom and Improved-CloCom.

(3) **Comment Extraction**: Once it has a list of useful code clones, CloCom retrieves the code comments from the AST of the database code segment and map the code comments to the code segments from the target project.
(4) **Comment Selection**: CloCom prunes out code comments which contain invalid information. That is, CloCom computed text similarity score between the comment and the cloned code snippets to measure the closeness of the code comment against the matched statements. It selects comments that have higher similarity scores.
(5) **Comment Ranking**: Finally, CloCom ranks the comments based on the text similarity score and the conciseness of the comments, and selects the best descriptions of the code segment from the list of available candidates.

In the comment extraction step, CloCom retrieves a code comment from the AST of cloned code segment from the database, and regards the comment as the description to the two similar code segments from the database and the target project. However, it is incorrect to directly map the comment to the cloned code snippets and reuse it because the scope of the comment is not perfectly fit to the cloned code snippets. For example, if the scope of the comment in database covers a superset of cloned statements, it can not be reused to describe the similar code snippet because it may contain some information about the statements outside the scope of the comment.

Therefore, we tried to combine our comment scope detection method to the comment extraction step and reimplement CloCom. However, the database of 1005 projects lacks details and accesses, so we could not reimplement the tool. Thus we applied our method after the last comment ranking step to filter out bad comments in CloCom. Specifically, we implemented Improved-CloCom as illustrated in Fig. 13, which utilized our method to detect the scopes of reused comments from the database and filtered out comments whose scopes are inconsistent with the cloned code snippets. Finally, we compared the ratio of good comments between results of CloCom and Improved-CloCom.

4.3.2. Data preparation

Based on the comment ranking result, we require source code repositories that the selected comments belong to for comment scope detection. Although Clocom has a large database of 1005 projects, it only extracted comments from a part of projects. Since CloCom has reused comments from 103 open source projects, we collected these open source Java projects from a repository host, GitHub, to build the database for comment scope detection. Our script queries the repository hosts search API and fetches all the hosted software. We combined CloCom with our comment scope detection method to generate comments for 21 Java open-source projects. The list of 21 projects is shown at Table 12.

Table 12
The results of CloCom and Improved-CloCom.

Project	*CloCom*				*Improved-CloCom*			
	Good	*Fix*	*Bad*	*Ratio*	*Good*	*Fix*	*Bad*	*Ratio*
Java JDK	23	5	68	24.0%	22	3	44	31.9%
ArgoUML	9	1	12	40.9%	9	0	5	64.3%
DNSJava	2	4	6	16.7%	2	4	2	25.0%
Ant	6	2	26	17.7%	4	1	17	19.1%
ANTLR	2	2	7	18.2%	2	2	4	25.0%
Carol	0	0	3	0.0%	0	0	2	0.0%
GanttProject	0	0	4	0.0%	0	0	3	0.0%
Hibernate	3	2	7	25.0%	2	1	3	33.3%
HSQLDB	8	3	24	22.9%	8	3	11	36.4%
JabRef	8	0	12	40.0%	8	0	8	50.0%
Jajuk	3	0	5	37.5%	3	0	3	50.0%
JavaHMO	0	1	4	0.0%	0	1	2	0.0%
JBidWatcher	1	1	14	6.3%	1	1	7	11.1%
JFtp	0	0	2	0.0%	0	0	1	0.0%
JHotDraw	6	0	1	85.7%	6	0	1	85.7%
MegaMek	1	0	12	7.7%	0	0	7	0.0%
Planeta	0	0	1	0.0%	0	0	1	0.0%
SweetHome	4	0	11	26.7%	3	0	9	25.0%
Vuze	1	0	18	5.3%	1	0	11	8.3%
FreeMind	3	0	6	33.3%	3	0	2	60.0%
FreeCol	5	0	10	33.3%	4	0	7	36.4%
TOTAL	85	21	253	23.7%	78	16	149	32.1%

4.3.3. Result and analysis

RQ6: Does our method contribute to automatic comment generation when mining existing source code from open source projects?

To evaluate the quality of the generated comments, Wong et al. performed a manual verification to evaluate the quality of the automatically generated code comments (Wong et al., 2015). The generated comment is good if it is accurate, adequate, concise and useful at describing the source code; or fix as it is not good but can be fixed with minor modifications; or bad. The result of Clocom is available at http://asset.uwaterloo.ca/clocom/.

Table 12 shows the results of Clocom and Improved-CloCom. CloCom generated code comments with a low yield and accuracy. It only generated 359 code comments for the 21 evaluated projects. Among them, there were 253 bad comments and only 23.7% of all the generated code comments can be directly applied to the code segments. After combining our comment scope detection method, Improved-CloCom filtered out 103 bad comments, and improved the ratio of good comments to 32.1%. Although Improved-CloCom has discarded a few good comments, it can significantly improve the efficiency of CloCom.

The reason that our comment scope detection method filtered out good comments can be summarized into three points. First, the scope of a comment detected by our method may cover a superset of the cloned code snippets. Because the comment has a de-

A Code Snippet From The Database

```
1 // get the internal data
2 int w = image1.getWidth();
3 int h = image2.getHeight();
4 int[] argb1 = new int[w*h];
5 int[] argb2 = new int[w*h];
6 image1.getRGB(0, 0, w, h, argb1, 0, w);
7 image2.getRGB(0, 0, w, h, argb2, 0, w);
```

A Code Snippet From A Target Project

```
1 int w = image1.getWidth();
2 int h = image2.getHeight();
3 int[] pixels = new int[w*h];
4 r.getPixels(0, 0, w, h, pixels)
```

Fig. 14. An example of good comments generated by CloCom and discarded by Improved-CloCom (The cloned code snippets are highlighted).

scription for the entire block of code other than the cloned code snippet, it can not be simply reused to describe the similar cloned code fragment from the target project. For example, at the top of Fig. 14, there is a piece of code segment from the database, which has a code comment at line 1. We have another code segment from the input project at the bottom of Fig. 14. They are matched after the code clone detection and the detected cloned code snippets are highlighted. Our comment scope detection method showed that the scope of the comment is line 2 to line 7, which is the superset of the cloned code snippets. Specifically, the two similar highlighted cloned code snippets are only to declare a few variables other than "get the internal data". Therefore, although it is appropriate for the entire code segment from the target project, the comment can not be reused to describe the highlighted cloned code snippet and our Improved-Clocom has discarded it.

Second, the scopes of a few good comments detected by our method may cover the subset of the cloned code snippets. If the scope of the comment in the database covers a subset of the cloned statements, it is inappropriate to reused the comment for the similar code snippet because it may be too simple for the entire block of code.

Third, our comment scope detection method has limitations in some specific scenarios. For example, if the scope of a comment covers the preceding or out-of-block statements, our method can not work well, which also results in inaccuracy of the comment scope detection method. However, these rarely happen and we only discovered 3 examples out of 359 comments.

Answer to RQ6: *Our method can significantly improve the efficiency of CloCom, which indicates that our method contributes to previous approaches on automatic comment generation when mining existing source code from open source projects.*

5. Related work

We group the related work into two categories: comment-code relationship analysis and general comment analysis.

5.1. Comment-code relationship analysis

Comments and codes are closely related, and many works studied their relationships from different aspects. Tan et al. (2012) proposed a tool called @tComment to test for outdated Javadoc comments. @tComment employs the Randoop tool to test whether the method properties (regarding null values and related exceptions) violate some constraints contained in Javadoc comments. Their automatic comment analysis technique achieved a high accuracy largely because Javadoc comments are well structured. In our work, we only focus on block and line comments which are not well structured in source code.

Tan et al. (2007) applied natural language processing (NLP) techniques to analyze code comments. They extracted the programmers' assumptions and requirements (referred to as comment rules) from the comments, and then performed a flow-sensitive and context-sensitive program analysis to check for mismatches between the comment rules and source code. Although their method achieved a high accuracy in detecting topic specific comments, it cannot be applied to arbitrary comments.

McBurney and McMillan (2016) conducted an empirical study examining method comments of source code written by authors, readers, and automatic source code summarization tools. Their work discovered that the accuracy of a human written method comment could be estimated by the textual similarity of that method comment to the source code, addressing that a good comment written by developers should have a high semantic similarity to source code.

Malik et al. (2008) conducted a large empirical study to better understand the rationale for updating comments in four large open source projects written in C. They investigated the rationale for updating comments along three dimensions: characteristics of the changed function, characteristics of the change itself, and time and code ownership characteristics.

Fluri et al. (2007) studied how comments and source code co-evolved over time. Their investigation results showed that 97% of comment changes are done in the same revision as the associated source code change. In their later paper (Fluri et al., 2009), eight different software systems were analyzed, with the finding that code and comments co-evolved in 90% of the cases in six out of the eight systems. They tried to map source code entities to comments using a token-based measure. However, it is insufficient to measure the textual similarity based on common key words appearing in both comments and code snippets.

5.2. Comment analysis

High quality source code comments play a crucial role in program comprehension and maintenance. Recently, many researchers have paid their attention on code comment quality evaluation and have proposed relevant evaluation metrics and tools. To the best of our knowledge, research on the problem of evaluating code comment quality was first reported in the 1990s (Oman and Hagemeister, 1992; Garcia and Granja-Alvarez, 1996). Oman and Hagemeister (1992) and Garcia and Granja-Alvarez (1996) evaluated code

comment quality by assessing the proportion of code comments in a software system. However, this evaluation metric is crude, as some redundant comments (e.g., copyright comments) were also considered.

Arafat and Riehle (2009) computed the density of comments in open source software code to find out how and why open source software creates high quality, and maintains this quality for ever larger project size. As a result, they found that successful open source projects follow a consistent practice of documenting their source code. Furthermore, their findings showed that comment density is independent of teams and projects.

Khamis et al. (2010) proposed a tool called JavadocMiner to evaluate the quality of code comments. JavadocMiner assessed code comment quality through a series of simple heuristic algorithms from both the association between code and comments and the language used for comments.

Haouari et al. (2011) presented an empirical study which analyzed existing comments in different open source Java projects. Their study showed that comments are intended to explain the code that follows them in the majority of cases. However, when to determine the relevance of the comments, they simply chose the following 20 lines of code to validate the relationship between comments and code.

Steidl et al. (2013) presented an approach for comment quality analysis and assessment, which was based on different comment categories using machine learning on Java and C/C++ programs. Their comprehensive quality model comprised quality attributes for each comment category based on four criteria including coherence between comments and source code. However, they only measured the coherence between the member comment and the method name. When dealing with inline comments, they only used the length of inline comments as an indicator of their coherence to the following lines of code.

Another comment practice concerns code comment generation. Many automatic comment generation methods and tools involve matching between natural language like comments and programming language by mining software repositories. Wong et al. (2013) proposed a novel method to automatically generate code comments by mining large-scale Q&A data from StackOverflow.[12] Wong et al. (2015) also proposed a general tool called CloCom to generate code comments automatically by analyzing existing software repositories. In addition, Sridhara et al. (2010) presented a novel technique to automatically generate descriptive comments for Java methods. Their method generated a comment that summarized the overall actions of the method. Moreno et al. (2013) presented a technique to automatically generate readable comments for Java classes. McBurney and McMillan (2014) presented a novel approach for automatically generating summaries of Java methods that summarize the context surrounding a method. Furthermore, Iyer et al. (2016) used Long Short Term Memory (LSTM) networks with attention to produce sentences that can describe C# code snippets and SQL queries.

6. Threats to validity

There are several threats that may potentially affect the validity of our experiments. Threat to external validity is that our study is restricted to Java open source projects. Thus, we cannot claim that our method can be applied to systems which are not implemented in Java. Threats to internal validity relate to our experiment data and algorithms we used. Although the dataset contains 4000 comment-statement pairs, it may be not enough to build a general model. Moreover, there are many strategies to determine the scope of a comment based on the statement categorization and our strategy to determine the scope of comments at the end may not be the best one. In Study II, although we have filtered out a number of noise data, threats to internal validity are also related to some errors in our experiment data. Because the dataset was extracted from the source code automatically, there was a large quantity of data without verification as a result of the overwhelming size of the dataset. In Study III, owing to the lack of details of the database and the settings of manual validation for the generated comments, we could not reimplement CloCom and have to reuse the experimental result of CloCom in their paper.

[12] https://stackoverflow.com/.

7. Conclusion

Comment plays an important role in program development and comprehension. However, it is seldom straight-forward to track relations between comments and source code entities algorithmically. This paper proposes a machine learning based approach for detecting scopes of source code comments automatically. By utilizing three dimensions of features, our method performed effectively on comment scope detection. Moreover, our method was applied to two tasks in software engineering. It improved the performance and efficiency of baseline methods in both tasks. As a general approach, our comment scope detection method would facilitate comment-code mapping for many automatic techniques in software engineering.

In the future, we intend to further combine the method with more automatic software artifacts (e.g., code search) which would utilize the information of comments and code snippets. More natural language processing techniques will be considered to enhance the evaluation of semantic similarities between comments and statements. Besides, we expect to employ learning-based approach to improve the accuracy in the detection of comment scopes.

Acknowledgments

This research is supported by the National Key R&D Program of China (2018YFB1004804), the National Natural Science Foundation of China (6167254), Science and Technology Planning Project of Guangzhou (201802020013), and China Postdoctoral Science Foundation (2018M640855).

References

Aggarwal, K.K., Singh, Y., Chhabra, J.K., 2002. An integrated measure of software maintainability. In: Reliability and Maintainability Symposium, 2002. Proceedings. Annual. IEEE, pp. 235–241.

Ambler, S.W., Vermeulen, A., Bumgardner, G., 1999. The elements of java style.

Arafat, O., Riehle, D., 2009. The commenting practice of open source. In: Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications. ACM, pp. 857–864.

Breiman, L., 2001. Random forests. Mach. Learn. 45 (1), 5–32.

Chen, H., Liu, Z., Chen, X., Zhou, F., Luo, X., 2018. Automatically detecting the scopes of source code comments. In: Computer Software and Applications Conference (COMPSAC), 2018 IEEE 42st Annual. IEEE. Accepted as a full paper.

Dit, B., Holtzhauer, A., Poshyvanyk, D., Kagdi, H., 2013. A dataset from change history to support evaluation of software maintenance tasks. In: Proceedings of the 10th Working Conference on Mining Software Repositories. IEEE Press, pp. 131–134.

DAmbros, M., Gall, H., Lanza, M., Pinzger, M., 2008. Analysing software repositories to understand software evolution. In: Software Evolution. Springer, pp. 37–67.

Fluri, B., Wursch, M., Gall, H.C., 2007. Do code and comments co-evolve? on the relation between source code and comment changes. In: Reverse Engineering, 2007. WCRE 2007. 14th Working Conference on. IEEE, pp. 70–79.

Fluri, B., Würsch, M., Giger, E., Gall, H.C., 2009. Analyzing the co-evolution of comments and source code. Softw. Qual. J. 17 (4), 367–394.

Gall, H.C., Fluri, B., Pinzger, M., 2009. Change analysis with Evolizer and Changedistiller. IEEE Softw. 26 (1), 26.

Garcia, M.J.B., Granja-Alvarez, J.C., 1996. Maintainability as a key factor in maintenance productivity: a case study. In: icsm, p. 87.

Genuer, R., Poggi, J.-M., Tuleau-Malot, C., 2010. Variable selection using random forests. Pattern Recognit. Lett. 31 (14), 2225–2236.

Goldberg, Y., Levy, O., 2014. Word2vec explained: deriving Mikolov et al.'s negative-sampling word-embedding method. arXiv: 1402.3722.
Haiduc, S., Bavota, G., Marcus, A., Oliveto, R., De Lucia, A., Menzies, T., 2013. Automatic query reformulations for text retrieval in software engineering. In: Proceedings of the 2013 International Conference on Software Engineering. IEEE Press, pp. 842–851.
Haouari, D., Sahraoui, H., Langlais, P., 2011. How good is your comment? A study of comments in java programs. In: Empirical Software Engineering and Measurement (ESEM), 2011 International Symposium on. IEEE, pp. 137–146.
Harris, Z.S., 1954. Distributional structure. Word 10 (2–3), 146–162.
He, H., Garcia, E.A., 2009. Learning from imbalanced data. IEEE Trans. Knowl. Data Eng. 21 (9), 1263–1284.
Howard, M.J., Gupta, S., Pollock, L., Vijay-Shanker, K., 2013. Automatically mining software-based, semantically-similar words from comment-code mappings. In: Proceedings of the 10th Working Conference on Mining Software Repositories. IEEE Press, pp. 377–386.
Huang, Y., Zheng, Q., Chen, X., Xiong, Y., Liu, Z., Luo, X., 2017a. Mining version control system for automatically generating commit comment. in: 2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), pp. 414–423.
Huang, Y., Chen, X., Liu, Z., Luo, X., Zheng, Z., 2017a. Using discriminative feature in software entities for relevance identification of code changes. J. Softw. 29 (7).
Huang, Y., Jia, N., Zhou, Q., Chen, X., Xiong, Y., Luo, X., 2018a. Guiding developers to make informative commenting decisions in source code, in: Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, ICSE '18. ACM, New York, NY, USA, pp. 260–261.
Huang, Y., Jia, N., Chen, X., Hong, K., Zheng, Z., 2018b. Salient-class location: Help developers understand code change in code review, in: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2018. ACM, New York, NY, USA, pp. 770–774.
Iyer, S., Konstas, I., Cheung, A., Zettlemoyer, L., 2016. Summarizing source code using a neural attention model. In: Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), vol. 1, pp. 2073–2083.
Kaelbling, M.J., 1988. Programming languages should not have comment statements. ACM Sigplan Not. 23 (10), 59–60.
Khamis, N., Witte, R., Rilling, J., 2010. Automatic quality assessment of source code comments: the JavadocMiner. In: International Conference on Application of Natural Language to Information Systems. Springer, pp. 68–79.
Ko, A.J., Myers, B.A., Coblenz, M.J., Aung, H.H., 2006. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. IEEE Trans. Softw. Eng. 32 (12).
Liaw, A., Wiener, M., et al., 2002. Classification and regression by randomforest. R news 2 (3), 18–22.
Liu, Z., Chen, H., Huang, Y., Chen, X., Luo, X., Zhou, F., 2018. Automatic detection of outdated comments during code changes. In: Computer Software and Applications Conference (COMPSAC), 2018 IEEE 42st Annual. IEEE. Accepted as a full paper.
Malik, H., Chowdhury, I., Tsou, H.-M., Jiang, Z.M., Hassan, A.E., 2008. Understanding the rationale for updating a functions comment. In: Software Maintenance, 2008. ICSM 2008. IEEE International Conference on. IEEE, pp. 167–176.
Marcus, A., Antoniol, G., 2012. On the use of text retrieval techniques in software engineering. In: Proceedings of 34th IEEE/ACM International Conference on Software Engineering, Technical Briefing.
McBurney, P.W., McMillan, C., 2014. Automatic documentation generation via source code summarization of method context. In: Proceedings of the 22nd International Conference on Program Comprehension. ACM, pp. 279–290.
McBurney, P.W., McMillan, C., 2016. An empirical study of the textual similarity between source code and source code summaries. Empir. Softw. Eng. 21 (1), 17–42.
Mikolov, T., Chen, K., Corrado, G., Dean, J., 2013. Efficient estimation of word representations in vector space. arXiv: 1301.3781.
Mikolov, T., Sutskever, I., Chen, K., Corrado, G.S., Dean, J., 2013. Distributed representations of words and phrases and their compositionality. In: Advances in Neural Information Processing Systems, pp. 3111–3119.
Miller, G.A., Beckwith, R., Fellbaum, C., Gross, D., Miller, K.J., 1990. Introduction to WordNet: an on-line lexical database. Int. J. Lexicog. 3 (4), 235–244.
Moreno, L., Aponte, J., Sridhara, G., Marcus, A., Pollock, L., Vijay-Shanker, K., 2013. Automatic generation of natural language summaries for java classes. In: Program Comprehension (ICPC), 2013 IEEE 21st International Conference on. IEEE, pp. 23–32.
Neamtiu, I., Foster, J.S., Hicks, M., 2005. Understanding source code evolution using abstract syntax tree matching. ACM SIGSOFT Softw. Eng. Notes 30 (4), 1–5.
Oman, P., Hagemeister, J., 1992. Metrics for assessing a software system's maintainability. In: Software Maintenance, 1992. Proceerdings., Conference on. IEEE, pp. 337–344.
Pascarella, L., Bacchelli, A., 2017. Classifying code comments in java open-source software systems. In: Proceedings of the 14th International Conference on Mining Software Repositories. IEEE Press, pp. 227–237.
Rong, X., 2014. Word2vec parameter learning explained. arXiv: 1411.2738
Schütze, H., 2008. Introduction to information retrieval. In: Proceedings of the International Communication of Association for Computing Machinery Conference.
de Souza, S.C.B., Anquetil, N., de Oliveira, K.M., 2005. A study of the documentation essential to software maintenance. In: Proceedings of the 23rd Annual International Conference on Design of Communication: Documenting & Designing for Pervasive Information. ACM, pp. 68–75.
Sridhara, G., Hill, E., Muppaneni, D., Pollock, L., Vijay-Shanker, K., 2010. Towards automatically generating summary comments for java methods. In: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering. ACM, pp. 43–52.
Steidl, D., Hummel, B., Juergens, E., 2013. Quality analysis of source code comments. In: Program Comprehension (ICPC), 2013 IEEE 21st International Conference on. IEEE, pp. 83–92.
Tan, L., Yuan, D., Krishna, G., Zhou, Y., 2007. /* iComment: bugs or bad comments?*. In: ACM SIGOPS Operating Systems Review, vol. 41. ACM, pp. 145–158.
Tan, S.H., Marinov, D., Tan, L., Leavens, G.T., 2012. @tComment: testing javadoc comments to detect comment-code inconsistencies. In: Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on. IEEE, pp. 260–269.
Thomas, S.W., Adams, B., Hassan, A.E., Blostein, D., 2014. Studying software evolution using topic models. Sci. Comput. Program. 80, 457–479.
Vermeulen, A., 2000. The Elements of Java (TM) Style, vol. 15. Cambridge University Press.
Wong, E., Liu, T., Tan, L., 2015. Clocom: Mining existing source code for automatic comment generation. In: Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on. IEEE, pp. 380–389.
Wong, E., Yang, J., Tan, L., 2013. AutoComment: mining question and answer sites for automatic comment generation. In: Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on. IEEE, pp. 562–567.
Woodfield, S.N., Dunsmore, H.E., Shen, V.Y., 1981. The effect of modularization and comments on program comprehension. In: Proceedings of the 5th International Conference on Software Engineering. IEEE Press, pp. 215–223.
Yang, J., Tan, L., 2012. Inferring semantically related words from software context. In: Mining Software Repositories (MSR), 2012 9th IEEE Working Conference on. IEEE, pp. 161–170.
Ye, X., Shen, H., Ma, X., Bunescu, R., Liu, C., 2016. From word embeddings to document similarities for improved information retrieval in software engineering. In: Proceedings of the 38th International Conference on Software Engineering. ACM, pp. 404–415.
Zheng, W., Zhou, H.-Y., Li, M., Wu, J., 2017. Code attention: translating code to comments by exploiting domain features. arXiv: 1709.07642.

Huanchao Chen is a postgraduate student at the Sun Yat-sen University. His research interest includes code analysis and comprehension, and mining software repositories.

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

Zhiyong Liu is a postgraduate student at the Sun Yat-sen University. His research interest includes software engineering, code analysis and comprehension.

Xiangping Chen is an associate professor in the School of Communication and Design, Sun Yat-sen University. She got her Ph.D. degree from the Peking University in 2010. Her research interest includes software engineering and mining software repositories.

Fan Zou is a professor in the School of Data and Computer Science, Sun Yat-sen University. His research interest includes smart home, software engineering and image processing.

Xiaonan Luo is a professor of School of Computer Science and Information Security, Guilin University of Electronic Technology. His research interests include image processing, computer graphics & CAD, mobile computing.