Automatic Detection of Outdated Comments During Code Changes

Zhiyong Liu[1], Huanchao Chen[1], Xiangping Chen[2,*], Xiaonan Luo[3], Fan Zhou[1]
[1]School of Data and Computer Science, National Engineering Research Center of Digital Life, Sun Yat-sen University, Guangzhou, China
[2]Institute of Advanced Technology, Sun Yat-sen University, Guangzhou, China
[3]School of Computer Science and Information Security, Guilin University of Electronic Technology, Guilin, China
Email: {chenhch6,liuzhy48}@mail2.sysu.edu.cn {chenxp8,isszf}@mail.sysu.edu.cn luoxn@guet.edu.cn

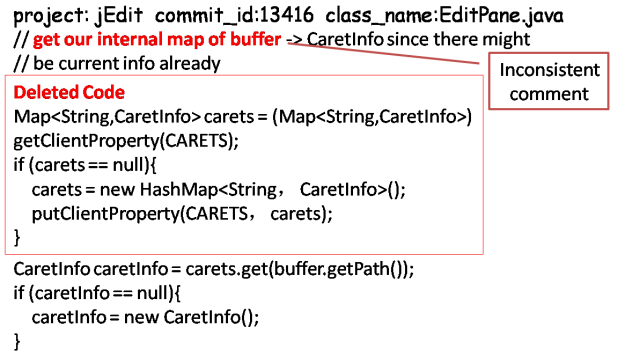
***Abstract*—Comments are used as standard practice in software development to increase the readability of code and to express programmers' intentions in a more explicit manner. Nevertheless, keeping comments up-to-date is often neglected for programmers. In this paper, we proposed a machine learning based method for detecting the comments that should be changed during code changes. We utilized 64 features, taking the code before and after changes, comments and the relationship between the code and comments into account. Experimental results show that 74.6% of outdated comments can be detected using our method, and 77.2% of our detected outdated comments are real comments which require to be updated. In addition, the experimental results indicate that our model can help developers to discover outdated comments in historical versions of existing projects.**

***Index Terms*—Outdated Comment Detection, Random Forest, Code Changes**

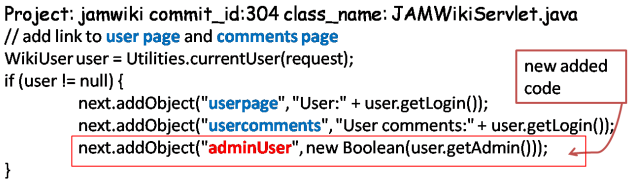
I. Introduction

Comments are used as standard practice in software development to increase the readability of code and express programmers' intentions in a more explicit manner. In practice, comments are the second most-used documented artifact for code understanding, with the first being the code itself [1]. The quality of comments is considered significant in software quality analysis [2]. But, as the example of Lakhotia shows, sometimes programmers do not care that someone else might want to understand the source code [3]. For software development and maintenance, the lack of comments as well as outdated comments are counter-productive. An outdated comment indicates either a bug or a bad comment both of which have severe implications for the software robustness and productivity[4]. Outdated comments may cause programmers to waste time and make code reviews more complicated [5]. Moreover, it often results in difficult-to-diagnose errors after release [6].

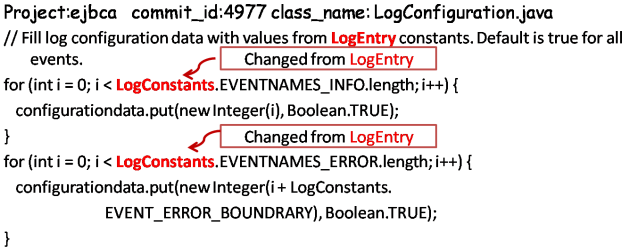
Most of the outdated comments were appeared after the code changes [7]. Figure 1 shows three outdated comment examples from open source projects. We can see that the comment in (a) includes a description of a behavior that is not implemented after the change. The comment in (b) does not mention "link is added to adminUser" and may mislead the code reviewer about the implementation of the functionality. The comment in (c) mentions non-existent variables and may confuse code readers.



(a) Comment includes description of deleted functionality



(b) Added functionality is not mentioned in the comment



(c) Object mentioned in the comment is changed

Fig. 1: Examples of Outdated Comments

Current studies on the quality of source code comments have proposed approaches to analyze the relationship between code and comments [6–8] and have indicated the importance of maintaining consistencies between the code and

* Corresponding author.

0730-3157/18/$31.00 ©2018 IEEE
DOI 10.1109/COMPSAC.2018.00028

comments. Methods have been proposed to detect out-of-date doc comments [1, 4, 6, 9, 10]. A high accuracy and recall were achieved when detecting outdated doc comments [9, 10] because doc comments are well structured for analysis. However, for methods that considered either doc or block/line comments, the detection was performed at a function/method level [4, 6] or was topic specific [1, 4].

Our method focuses on the automatic detection of the outdated block/line comments during code changes. To detect outdated comments at a fine-grained level, our approach takes a comment as an evaluation unit, and utilizes machine learning technology to validate whether the comments should be changed. We utilized 64 features considering the code before and after changes, along with the comments and the relationship between the code and comments. Experimental results show that 74.6% of outdated comments can be detected using our method, and 77.2% of our detected outdated comments are real comments which require updating. In addition, the experimental result indicates that our model can effectively help developers to discover outdated comments in historical versions of existing projects.

II. Related Work

Source code comments play a crucial role in program comprehension and maintenance. A survey of software maintainers done by de Souza et al. [1] found that developers use comments as a key element to understand source code. It highlights the critical role of comments for software development and maintenance. Arafat et al. [11] looked at 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.

Comments are considered as documented knowledge for developers, and their quality is important in evaluating software quality. McBurney et al. [12] 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. Steidl et al. [2] 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: consistency throughout the project, completeness of the system documentation, coherence with the source code, and usefulness to the reader. Both aforementioned studies used one aspect about textual similarity to determine the inconsistency between the code and comments, which is insufficient for such a complicated problem. Aman et al. [13] conducted an empirical analysis on the usefulness of local variable names and comments in software quality assessments from the perspective of six popular open source software products. Their study showed that a method having longer named local variables is more change-prone. To understand if and how database-related statements are commented in source code, Linares-Vásquez et al. [14] mined Java open source projects that use JDBC for the data access layer from Github. They found that 77% of 33K+ source code methods do not have header comments. They also pointed out that existing comments were rarely updated when the related source code was modified.

Considering the importance of comments in programming practice, and the fact that an outdated comments may confuse and mislead programmers, several researchers [6–8] have investigated how code and comments co-evolve. Fluri et al. [7] conducted an empirical survey on three open source systems to study 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 Fluri's other paper [8], 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. Ibrahim et al. [6] believed that bug prediction models play important roles in the prioritization of testing and code inspection efforts. They studied comment update practices in three large open source systems written in C and Java and found that a change in which a function and its comment are suddenly updated inconsistently, has a high probability of introducing a bug.

There are three types of comments used in programming: doc, block, and line comments. Because doc comments are important for understanding an application programming interface (API), an outdated Javadoc comments can mislead method callers. Tan et al. [9] 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. Khamis et al. [10] examined the correlation between code quality and Javadoc comment-code inconsistencies. However, only some simple issues were checked, e.g., whether the parameter names, return types, and exceptions in the @param, @return and @throws tags were consistent with the actual parameter names, return types, and exceptions in the method, respectively. Their automatic comment analysis technique achieved a high accuracy largely because Javadoc comments are well structured, with few paraphrases and variants.

Some studies have considered both doc comments and block/line comments as the outer and inner comments of a function/method. Tan et al. [4] 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. Malik et al. [15] 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. Unlike our method, these two studies both used a method/function as the unit in detecting consistencies. TODO comments, which are used by developers to denote pending tasks, may lead to out-of-date comments when developers perform the mentioned tasks and then forget to remove the comments. Sridhara [16] presented a novel technique to automatically detect the status of a TODO comment using three aspects, including information retrieval, linguistics and semantics. The results showed that his status checker achieved a high accuracy, precision, and recall in checking whether a TODO comment was up-to-date.

Our method focuses on the detection of the block/line outdated comments during code changes. In contrast with existing techniques, our method is aimed at automatically detecting outdated comments at a fine-grained level for arbitrary comments.

III. Approach

A. Software Change Extraction

A software change is abstracted as *Change* = ⟨*Stmts, Stmts'*⟩, where *Stmts* and *Stmts'* are the sets of changed statements before and after a change. The set of statements is abstracted as *Stmts* = ⟨*Statements, Comments, Code Snippets*⟩. *Statements* are the code which are affected during a change. *Comments* are the set of comments whose scope covers a subset of the statements in *Statements*. *Code Snippets* are the code in the scope of *Comments*. For each comment belonging to *Comments*, the statements in its scope are included in the *Code Snippets*. Its scope covers a subset of the *Statements* where the change may cause the updating of the comment. We view the comments which were changed in software changes as outdated comments.

$$\forall \text{ comment} \in \textit{Comments}, \text{comment.scope} \subseteq \textit{CodeSnippet} \wedge \text{comment.scope} \cap \textit{Statements} \neq \emptyset$$

In this paper, we only focus on block and line comments because doc comments are well structured and their outdated comments detection problems have been well studied [9, 10]. Block and line comments are associated with several lines or only one line of code. In practice, block and line comments are used interchangeably. For example, consecutive line comments are used as a syntactical alternative to block comments. As a result, line comments were treated as a special kind of block comments.

Unlike doc comments, there are no definitions for the scopes of block and line comments. We used a set of heuristic rules to deduce the scope of a comment as follows:

(1) Ignore single-line comment, which contains code in front of the comment;
(2) Adjoining comments are considered to be one comment because there is no code between them;
(3) The scope of a comment starts from the first line of the code after the comment;
(4) The scope of a comment ends at the last line of code before another comment in the same block (not a sub-block), or the ending of the block, or the ending of a method.

It should be noted that using these rules to determine the scope of a comment may be not precise because a comment may only cover a subset of its following statements. However, the scope includes all the statements that may affect the comment updating. In the feature selection, features for the relationship between the code and comments are used to reduce possible inaccuracies.

After determined the scopes of the *Comments*, we extracted changed code between two versions using a tool called ChangeDistiller. ChangeDistiller is an implementation of a tree differencing algorithm for fine-grained source code change extraction, it enables fine-grained change type extraction and analysis to reason about coding conventions, control or exception flow, and even code and comment coevolution [17].

The source code changes in different classes or in the same class but different methods are separated as different *Changes*. For source code changes in the same method, they are considered separate *Changes* when the intersection of their comments' scopes is null. If the *Comments* of the *Stmts* in two *Changes* contain the same comment, these two *Changes* are then combined. This is because changed statements in the scope of the same comment cannot be used separately in evaluating the probability of updating the comment. As a result, a comment can only be included in the *Comments* of the *Stmts* of one *Change*.

$$if\ \textit{C1.Stmts.Comments} \cap \textit{C2.Stmts.Comments} \neq \emptyset, \textit{ then } \text{Combine}(\textit{C1,C2}) = \{\textit{C1.Stmts} \cup \textit{C2.Stmts}, \textit{C1.Stmts'} \cup \textit{C2.Stmts'}\}$$

It should be noted that because the *Comment*'s scope may change during the source code change, the statements in the *Stmts'* might not belong to the same *Comment*. As shown in Figure 2, the scope of the first *Comment* is reduced to the first two lines of code because our algorithm for detecting the *Comment*'s scope is not accurate. We do not separate the changed statements into two *Changes* because the first *Comment* may be affected if it contains a description of the deleted control statement.

In addition, when extracting *Comments* and *Code Snippets* based on the *Statements* of *Stmts'*, a comment may exist in the *Comments* of the *Stmts'* in various *Changes* because the

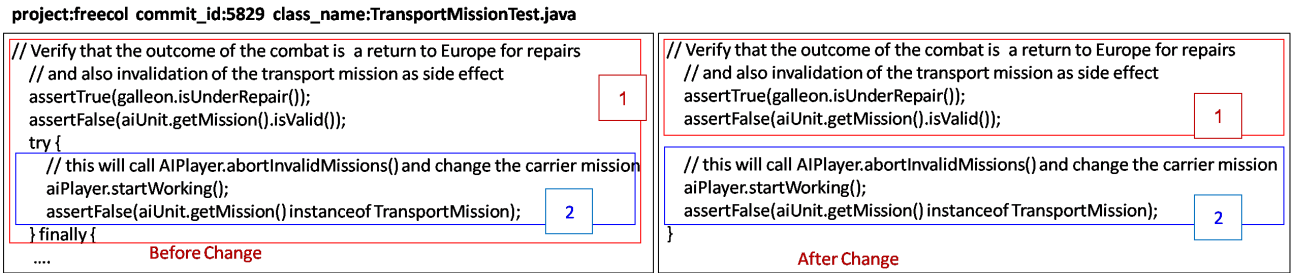


Fig. 2: The First Example of Extending the Comment's Scope

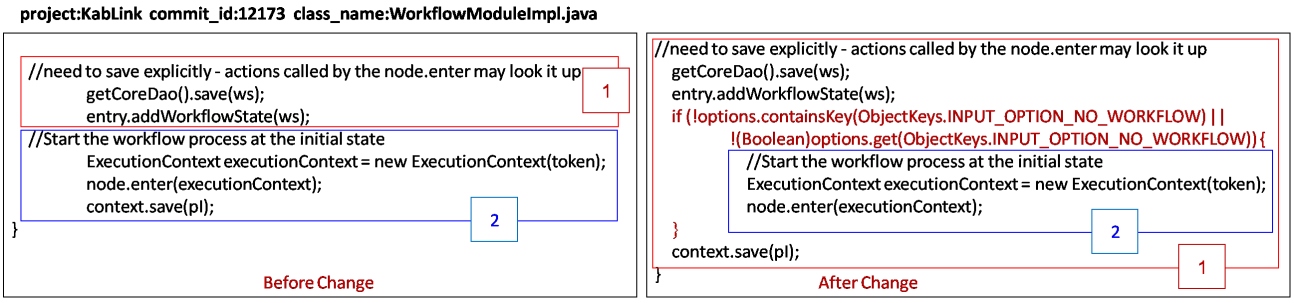


Fig. 3: The Second Example of Extending the Comment's Scope

comments' scope changes during the source code changes. As shown in Figure 3, the scope of the first comment extends to include more statements because a new "if" statement is added. Because the added statement may affect the possible updating of the first comment, we combine these kinds of *Changes*.

$if\ C1.Smts'.Comments \cap C2.Smts'.Comments \neq \emptyset, then$ Combine($C1, C2$)

B. Feature Selection

This section introduces the method used to select features from the code and comments. The comments whether required updating can be determined by the status of code changes, comments, and the relationship between the comments and code. We selected 64 features from three dimensions: the code, comments, and the relationship between the code and comments.

1) Code Features: Code features are used to describe the context of the changes, as well as the status of the code before and after the changes. We extract features at the class and method levels to describe the context, and utilize statement level information for the changes. Table I shows the feature list.

In programs written in object oriented programming languages, the class and method declarations are in the context of the code change. In the class where the code change is located, if class attributes are changed and the changed class attributes are used in the changed code, the change in an attribute may affect the functionality implementation, and the comment should be changed accordingly. Similarly, in the method where the changed code is located, if the return type or parameters of the method are changed, and the return statements or changed

TABLE I: Code Features

Level	*Name*	*Description*
Class Level	Changes on class attributes	Whether there are changes on the class attributes?
	Class attribute related	Are there changed class attribute(s) used in the code snippet?
Method Level	Changes on method declaration	Whether there are changes on the method declaration: method name, return type, and parameter list?
	Input/output related	Whether the code snippet contains return statements or usage of changed method parameters?
Statement Level	Number of statements	The number of statements in the code snippet
	Number of changed statements	The number of changed statements in the code snippet
	Percentage of changed statements	The percentage change statements in the code snippet
	Statement changes	12 types of statement changes in the code snippet: *IF* statement, *ELSE IF* statement, *FOR* statement, *Enhanced FOR* statement, *WHILE* statement, *CATCH* statement, *TRY* statement, *THROW* statement, *PRINT* statement, *LOG* statement, Method call statement, and Variable Declaration; 3 change types: add, delete, and modify.Total is 36 (12×3).
	Refactorings	Whether the change includes code refactorings (9 types of refactoring)

parameters are used in the changed code, the corresponding comments need to be updated.

In addition, the number of changed statements has a certain impact on the comment changes. As listed in Table II, the proportion of comment changes increases with the number of changed statements. In other words, they are proportionate to each other. In the feature selection, we added the number and percentage of changed statements, as well as the number of statements in the code snippets, to the code features.

TABLE II: Statistics of Lengths of Code Changes

Number of changed statements	*1-3*	*4-6*	*7-9*	*10-12*	*13-15*	*>15*
Comment changed	1,891	866	536	382	282	762
Comment unchanged	23,933	3,709	1,287	593	283	526
Percentage of change	7.3%	18.9%	29.4%	39.1%	49.9%	59.2%

Previous studies did not investigate how different types of statement changes affect the updating of comments. Based on the commit data collected from five projects: JEdit, Opennms, JAMWiki, EJBCA, JHotDraw (the data was also used in the experiments), we explored the relationship between the number of different types of changed statements and the number of updates in the associated comments, as listed in Table III. We can see that the impacts of different types of statement changes on the comment updating are quite different. Besides adding or deleting a *WHILE* statement, deleting an *ELSE IF*, an *Enhanced FOR*, or a *TRY* statement produces a probability of more than 45% that the comments will require modification. Moreover, adding a *FOR* statement, as well as deleting *CATCH* and *THROW* statements produces a probability of more than 40% that the comments tend to be updated. This indicates that these listed *Changes* are more relevant to comment updating. Adding or deleting a statement has a much bigger effect than modifying a statement on the updating of a comment. As a result, we added the 36 types of statement changes to the statement level features.

When the code change involves refactoring, other code features are not important in determining whether the comment should be updated. Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code, yet improves the modular structure of the software [18]. Code refactoring generally involves a large range of code modifications. Nevertheless, the modified code would not change the behavior of the function. The associated comment may be kept unchanged if the comment describes the behavior without details. Therefore, in the feature selection, we selected nine types of code refactoring features, as listed in Table IV.

2) Comment Features: Table V lists the comment features. The length of the comment affects the effectiveness and precision of the description [8]. A comment containing more detailed descriptions is more inclined to be updated. In addition, considering the context features, we added the ratios of the comments to the code of code snippets, methods, and classes respectively in the comment features.

In addition, some special keywords are used in comments to indicate that the code and comments need to be updated in the next revision, such as the task annotations ("TODO", "FIXME", "XXX") and bug tokens ("bug"). When updating code with these kinds of comments, the programmers are supposed to update the corresponding comments. For example, when a new function implementation is added in a position as expected, the term "TODO" in the original comment should be deleted so as to avoid confusion [19].

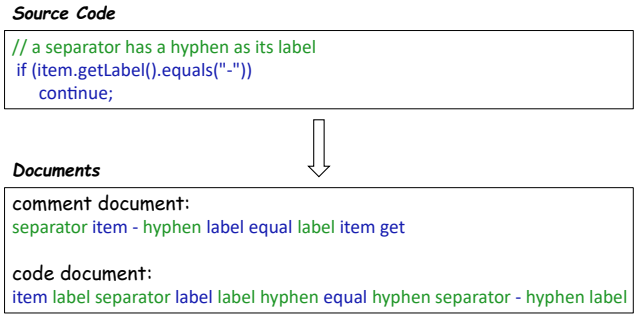


Fig. 4: A Document Generation Example

3) Relationship Features: There are divergent guidelines for how to write useful comments [2, 12]. Comments are supposed to contain the intentions and goals of the implementation, and possible additional insights behind the implementation [2]. As a result, the description may mention the object or the functionality in the code snippet. 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 [12]. If the similarity between the code and its corresponding comment changes significantly after the code change, the comment is likely to be modified. However, computing the similarity between the code and comments is difficult because there is a 'lexical gap' between programming language and natural language. In such cases, the similarity cannot be accurately measured by counting the common words contained in the code and comments or by word embedding directly. To overcome this issue, we developed a Skip-gram model based on the method proposed by Xin et al. [20], which bridges the lexical gap by projecting natural language statements and code snippets as meaning vectors in a shared representation space.

First, we preprocessed the source code and comments including word splitting, stopping word removal and word stemming. Second, for each word in the comment, we randomly selected two words in the code snippet and added them to the comment as a comment document. Third, for the code snippet, we randomly selected two words from the associated comment and added them to the code document. Finally, these two documents were merged and served as the corpus of our

TABLE III: Statistics of Statement Change Types

	Comment	*If*	*Else If*	*For*	*Enhanced-For*	*While*	*Catch*	*Try*	*Throw*	*Print*	*Log*	*Method Call*	*Variable Declaration*	*Average*
Insert	Changed	1,752	212	147	121	172	596	432	360	72	840	3,067	1,690	–
	Unchanged	3,404	379	219	332	206	991	716	658	206	1,994	8,563	3,663	–
	Changed percentage	34%	36%	40%	27%	46%	38%	38%	35%	26%	30%	26%	32%	33.72%
Modify	Changed	868	180	71	40	109	238	164	240	42	611	2,299	1,379	–
	Unchanged	3,321	550	203	173	177	740	389	903	279	3,289	20,254	8,694	–
	Changed percentage	21%	25%	26%	19%	38%	24%	30%	21%	13%	16%	10%	14%	20.58%
Delete	Changed	1,071	153	118	55	173	472	330	290	43	615	1,944	1,192	–
	Unchanged	1,812	166	240	66	214	719	392	439	162	1,123	5,300	2,514	–
	Changed percentage	37%	48%	33%	45%	45%	40%	46%	40%	21%	35%	27%	32%	36.50%

TABLE IV: Detected Refactoring Types

ID	*Refactoring*	*Description*
1	Extract Method	Turn the fragment into a method whose name explains the purpose of the method.
2	Inline Method	Put the method's body into the body of its callers and remove the method.
3	Rename Method	Change the name of the method.
4	Add Parameter	Add a parameter for an object that can pass on this information.
5	Remove Parameter	Remove a parameter which is no longer used by the method body.
6	Inline Temp	A temp that is assigned to once with a simple expression. Replace all references to that temp with the expression.
7	Encapsulate Field	There is a public field. Make it private and provide accessors.
8	Introduce Assertion	A section of code assumes something about the state of the program. Make the assumption explicit with an assertion.
9	Replace Exception With Test	When throw an exception on a condition the caller could have checked first, change the caller to make the test first.

TABLE V: Comment Features

Name	*Description*
Length of the comment	Split the comment to a word list, and count the number of words in the list
Task comment	Does the comment contain task annotations ("TODO", "FIXME", "XXX")
Bug or version comment	Does the comment contain "bug" or version annotation("bug", "fixed bug", "version")
The ratio of comment lines to the class	The ratio of lines of comment to lines of code in the class
The ratio of comment lines to the method	The ratio of lines of comment to lines of code in the method
The ratio of comment lines to the code snippet	The ratio of lines of comment to lines of code in the code snippet

Skip-gram model. An example is shown in Figure 4. Using the documents generation rules, we generated two documents from the code snippet and its comment. The green words were generated by the comment, and the blue words came from the code snippet.

To obtain the vector representation of each word, we then trained our Skip-gram model based on the corpus. The continuous skip-gram is effective at predicting the surrounding words in a context window of $2k+1$ words (we set $k=2$ and the window size is 5). The objective function of the skip-gram model aims at maximizing the sum of the log probabilities of the surrounding context words conditioned on the central word [21]:

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

where w_i and w_{i+j} respectively denote the central word and the surrounding context words within a context window of length $2k+1$. n denotes the length of the word sequence. The term $\log p(w_{i+j}|w_i)$ is the conditional probability, which is defined using the softmax function:

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

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 [21] to compute this probability.

To compute the comment and code similarity, we define three types of similarity measures:

(1) Word to word: Given two words w_1 and w_2, we define 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:

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

(3) Sentence to sentence: Between two sentences S_1 and S_2, we define their semantic similarity as follows:

$$sim(S_1, S_2) = \frac{sim(S_1 \to S_2) + sim(S_2 \to S_1)}{2} \tag{5}$$

where

$$sim(S_1 \rightarrow S_2) = \frac{\sum_{w \in S_1} sim(w, S_2)}{n} \quad (6)$$

where n denotes the number of words in S_1.

We considered the similarities between the comment and the code before and after the change. If the comment and the code before the change have a high similarity while they have a low similarity after the code is changed, there tends to be a large probability of modifying the comment to ensure the consistency of the comment and the code. Similarly, we computed the similarities between the comment and changed statements before and after changes. Moreover, we compared the difference between the comment and changed statement similarities before and after the change. Table VI shows all the relationship features.

TABLE VI: Relationship Features

Name	*Description*
The similarity of code and its comment before the change	The similarity between the code and the old comment before the change
The similarity of code and its comment after the change	The similarity between the code and the old comment after the change
The distance of code and comments similarities	The distance of similarities of code and old comments before and after the change
The similarity of changed statements and the comment before the change	The similarity of changed statements and its old comment before the change
The similarity of changed statements and the comment after the change	The similarity of changed statements and its old comment after the change
The distance of changed statement and comment similarities	The distance of the similarities of changed statements and their old comments before and after the change

C. Machine Learning

In this paper, we regard the problem of detecting outdated comments during code changes as a binary classification problem. Whether the comments were outdated could be determined by the code changes, the comments, and the relationship between the comments and code. By establishing the most likely features for a given category outdated comments or not outdated comments, we employed random forests [22], one of the supervised algorithms, to classify outdated comments and not outdated comments.

The Random Forests algorithm builds a large number of basic classifiers and let them vote for the most popular class. In our case study, we selected 100 Classification And Regression Trees (CART) as the basic classifiers and set the random subset value of the features as 8 (square root of all features). CART uses *Gini* function to choose the best splitter. For binary classification problem, the *Gini* score is calculated as follow:

$$Gini(D) = 1 - \left(\frac{|c_0|}{|D|}\right)^2 - \left(\frac{|c_1|}{|D|}\right)^2 \quad (7)$$

Where D is the set of samples, and $|D|$ is the number of samples in the set D, $|c_0|$ is the number of class 0 samples in D, and $|c_1|$ is the number of class 1 samples in D.

The gain of one splitter is calculated as follow:

$$Gain(D, A) = Gini(D) - \frac{|D_1|}{|D|}Gini(D_1) - \frac{|D_2|}{|D|}Gini(D_2) \quad (8)$$

Where A is a attribute which splitted D in two subsets D_1 and D_2. $|D_1|$ is the number of samples in the set D_1, and $|D_2|$ is the number of samples in the set D_2.

IV. Experiments

A. Data Collection

In order to build a data set to evaluate the effectiveness of our approach, we first collected the source code from five open source projects, as presented in Table VII. They are all rated above 4.8 (the perfect score is 5) in Sourceforge and are widely used in source code analysis research papers. They have different application domain types, including text editor, managing system, collaborative software, Wiki engine, and two-dimensional graphical framework.

From these five projects, we filtered out *Changes* with low quality comments whose contents (1) contained less than three words; (2) were out-of-date statements; (3) were symbols. Eventually, 35,050 *Changes* were collected from 9,909 commits. Among these *Changes*, the comments of 30,455 *Changes* were not changed, and 4,595 were changed, we treated these comments as outdated comments. We randomly selected 400 *Changes* in each project, for a total of 2,000. These 2,000 *Changes* were used to validate RQ2. And the remaining 33,050 *Changes* were used to train and evaluate our model in RQ1, as shown in Table VIII.

TABLE VII: Projects Used in the Experiments

Project	*Date*	*Commit*	*Change*	*Comment*	
				Changed	*Unchanged*
JEdit	2000/01-2016/03	2,133	4,750	662	4,088
OpenNMS	2002/05-2009/11	1,558	5,585	941	4,644
JAMWiki	2006/06-2013/03	1,262	3,756	467	3,289
EJBCA	2001/11-2017/01	4,705	19,800	2,442	17,358
JHotDraw	2000/10-2016/01	251	1,159	83	1,076
Total	–	9,909	35,050	4,595	30,455

TABLE VIII: Datasets in the Experiments

Dataset	*Change*	*Comment*	
		Changed	*Unchanged*
Dataset of RQ1	33,050	4,320	28,730
Dataset of RQ2	2,000	275	1,725

B. Research Question

This research primarily had the goal of detecting outdated comments during code changes based on automatic judgements. For this purpose, we focus on two questions:

RQ1: How effective is our method in predicting outdated comments during code changes?

In this work, our main purpose is to detect outdated comments during code changes. Therefore, we would like to further investigate whether our model can perform effectively in predicting outdated comments.

The automatic judgement for outdated comments was performed by a classifier trained by measurable features. By convention, we used three metrics, including *precision* rate, *recall* rate and *F1-score* to evaluate the performance of our method. Throughout the experiment, we labeled a code change with a comment change as "1" (outdated comment) and "0" for the code change without a comment change (not outdated comment).

RQ2: Is our method helpful in discovering outdated comments in historical versions of existing projects?

To determine how our method could be used to help programmers find out outdated comments in historical versions of existing projects, we applied it in the *Dataset of RQ2*. We invited 9 participants to validate the detected outdated comments which were not changed in existing projects.

C. Results

RQ1: How effective is our method in predicting outdated comments?

To answer RQ1, we performed experiments to confirm that the selected features are applicable in predicting outdated comments. First, we conducted three separate experiments on different dimensions. We trained three random forest classifiers by separately selecting the code features, the comment features and the relationship features. Then, on the basis of the classifier utilizing the code features, we added the remaining two dimensions of features to the model sequentially and conducted another two experiments. We used 10-fold cross-validation to evaluated our models and compared the *precision*, *recall* and *F1-Score* metrics of the five experiments. As listed in Table IX, among the three dimensions, the most effective and best features for the classification performance are the code features. The second are the relationship features, and the comment features play a smaller role in classification. After adding the relationship features to the code features, the *precision* rate of the positive instances increased from 83.6% to 86.8% and the *recall* rate increased from 58.8% to 60.6% as well. With the comment features, the *precision* rate and the *recall* rate increased from 86.8% to 88.4% and from 60.6% to 61.5% respectively.

In Table IX, we can see that our *recall* rates of positive instances is relatively low. One of the reasons is that there is a large disparity in the proportion of positive and negative instances in the data set, which is about 1:6.6. To improve the performance of classification decision trees and to obtain better models with such 'skewed data', we introduced the cost matrix in the classifier [23]. 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, $C_{00} = C_{11} = 0$. To introduce the cost matrix, we modified the calculation of *Gini* score as:

$$Gini(D) = 1 - \left(\frac{|c_0| \times C_{01}}{|D'|}\right)^2 - \left(\frac{|c_1| \times C_{10}}{|D'|}\right)^2 \quad (9)$$

where

$$|D'| = |c_0| \times C_{01} + |c_1| \times C_{10} \quad (10)$$

We adjusted the proportion of C_{01} and C_{10} to observe how the *precision* rate and *recall* rate of positive instances reacted which is illustrated in Figure 5. The curves indicate that the *precision* rate of positive instances increases as the proportion of C_{01} and C_{10} increases, whereas the *recall* rate decreases when the proportion of C_{01} and C_{10} increases. When the proportion of C_{01} and C_{10} is 1:6, the *precision* rate is 77.2% and the *recall* is 74.6%. At the same time, the *F1-score* achieves the highest score, 0.759. Therefore, we can adjust the proportion of C_{01} and C_{10} to affect the *precision* and *recall* rates of positive instances for specific applications.

RQ2: Is our method helpful in discovering outdated comments in historical versions of existing projects?

To answer RQ2, our goal was to help developers discover outdated comments which were not changed in the history versions of a project. While ensuring a high *recall* rate, we hoped to keep the *precision* rate as high as possible. To this end, we chose the cost matrix leading to the highest *F1-Score* in Figure 5, as follows:

$$\begin{bmatrix} 0 & 1 \\ 6 & 0 \end{bmatrix}$$

As shown in Table VIII, the *Dataset of RQ2* contains 2,000 *Changes*, the comments of 1,725 *Changes* were not changed,

TABLE IX: Results of the Five Experiments

Featrues	*Class 1 (Outdated Comment)*			*Class 0 (Not Outdated Comment)*		
	Pre-cision	*Recall*	*F1-Score*	*Pre-cision*	*Recall*	*F1-Score*
Code features	83.6%	58.8%	0.691	93.9%	98.2%	0.960
Comment features	47.3%	12.6%	0.199	87.8%	97.8%	0.925
Relation-ship features	66.5%	38.1%	0.381	91.0%	97.0%	0.939
Code and relation-ship features	86.8%	60.6%	0.713	94.1%	98.6%	0.956
All features	88.4%	61.5%	0.726	94.3%	98.7%	0.965

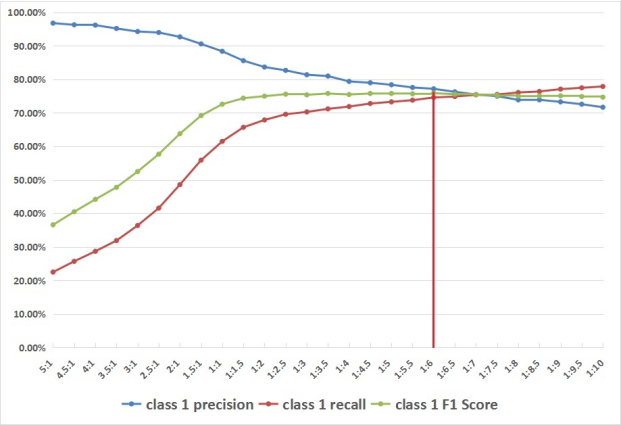


Fig. 5: Curves of Presicion Rate, Recall Rate and F1-score of Positive Instances for Different Cost Matrixs

and 275 were changed. Utilizing the model trained in RQ1, we obtained 279 *Changes* which have been classified as positive instances from the *Dataset of RQ2*. Among these, 210 comments have already been updated in the projects, while 69 comments remained unchanged. Then, we asked 9 participants to validate these 69 *Changes* on the website. Among the 9 participants, there are 6 graduate students and 3 PhD students. They all have Java development background, with an average development experience of 5 years. These 9 participants were divided into 3 groups and each group validated all the 69 *Changes*. The webpage is shown in Figure 6. A participant is given a view of one *Change* containing the code before and after the change as well as the comment before the change. In order to facilitate the verification, we highlighted the changed lines of the code. For each *Change*, participant has five options to select, as shown in Table X. In this way, we obtained three answers for each *Change*. The results are illustrated in Table XI.

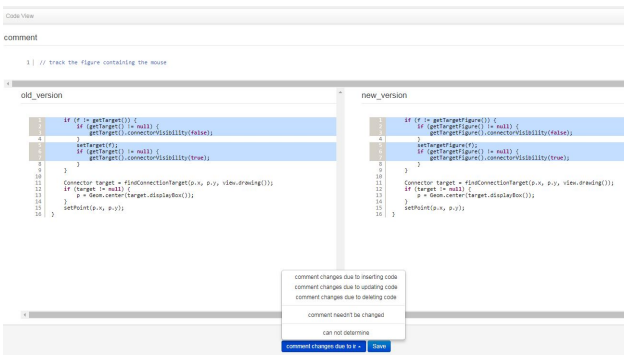


Fig. 6: The Webpage of Validation

As shown in Table XI, 31 (19+6+6) *Changes* were deemed to require to change their comments by more than or equal to two groups Of the 69 *Changes* (the last line in Table XI). This meant that our method found out 241 (210+31) outdated

TABLE X: Validation Options

ID	*Validation Option*
O_1	Comment changes due to inserting code
O_2	Comment changes due to updating code
O_3	Comment changes due to deleting code
O_4	Comment needn't be changed
O_5	It is hard to determine whether the comment should be changed

TABLE XI: Results of Validations

Group	O_1	O_2	O_3	O_4	O_5
Group 1	15	7	7	36	4
Group 2	17	6	5	33	8
Group 3	20	7	4	32	3
Synthesis	19	6	6	31	7

comments and more than 10% of these comments were forgot to change in existing projects. It indicated that our method can help developers to discover the outdated comments in historical versions of existing projects.

V. Threats to Validity

There are several threats that may potentially affect the validity of our experiments. There is a threat to the external validity because our study was 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 are related to the errors in our experiment data and the tools or algorithms we used. Because the data set 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 data set. The quality of data set was affected by several factors. First, outdated comments which were not changed existed in the data set. Second, in the software change extraction stage, the tool used to extract the code changes and the heuristic rules used to determine the scopes of the comments were not accurate under some conditions. In addition, our refactoring detection only covered a sub set of refactorings which are commonly used.

VI. Conclusion and Feature Work

This paper proposed a machine learning based approach for detecting outdated comments during code changes. The change in the source code and the code-comment relationship before and after the changes provide more clues for detecting possible comment changes. We utilized 64 features of the software change, as well as the status of comments and their relationship with the code before and after changes, and employed machine learning approach for detection.

In the future, the relationship between the changes in the same commit will be considered as the context of the change to enhance the detection accuracy. More natural language

processing techniques will be considered to enhance the evaluation of semantic similarity. In addition, we will endeavor to locate the line of code and the words in a comment that affect the change to provide more fine-grain detection.

ACKNOWLEDGMENT

This research is supported by the National Natural Science Foundation of China(61672545, 61502546), the National Key R&D Program of Chin2018YFB1004804, the Science and Technology Planning Project of Guangdong Province (2013B090700009), the Science and Technology Planning Project of Zhongshan(2016A1044).

REFERENCES

[1] S. C. B. de Souza, N. Anquetil, and K. M. de Oliveira, "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, 2005, pp. 68–75.

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

[3] A. Lakhotia, "Understanding someone else's code: Analysis of experiences," *Journal of Systems and Software*, vol. 23, no. 3, pp. 269–275, 1993.

[4] L. Tan, D. Yuan, G. Krishna, and Y. Zhou, "/* icomment: Bugs or bad comments?*," in *ACM SIGOPS Operating Systems Review*, vol. 41, no. 6. ACM, 2007, pp. 145–158.

[5] D. L. Parnas, "Precise documentation: The key to better software," in *The Future of Software Engineering*. Springer, 2011, pp. 125–148.

[6] W. M. Ibrahim, N. Bettenburg, B. Adams, and A. E. Hassan, "On the relationship between comment update practices and software bugs," *Journal of Systems and Software*, vol. 85, no. 10, pp. 2293–2304, 2012.

[7] B. Fluri, M. Wursch, and H. C. Gall, "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, 2007, pp. 70–79.

[8] B. Fluri, M. Würsch, E. Giger, and H. C. Gall, "Analyzing the co-evolution of comments and source code," *Software Quality Journal*, vol. 17, no. 4, pp. 367–394, 2009.

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

[10] N. Khamis, R. Witte, and J. Rilling, "Automatic quality assessment of source code comments: The javadocminer." in *NLDB*. Springer, 2010, pp. 68–79.

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

[12] P. W. McBurney and C. McMillan, "An empirical study of the textual similarity between source code and source code summaries," *Empirical Software Engineering*, vol. 21, no. 1, pp. 17–42, 2016.

[13] H. Aman, S. Amasaki, T. Sasaki, and M. Kawahara, "Empirical analysis of change-proneness in methods having local variables with long names and comments," in *Empirical Software Engineering and Measurement (ESEM), 2015 ACM/IEEE International Symposium on*. IEEE, 2015, pp. 1–4.

[14] M. Linares-Vásquez, B. Li, C. Vendome, and D. Poshyvanyk, "How do developers document database usages in source code?(n)," in *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*. IEEE, 2015, pp. 36–41.

[15] H. Malik, I. Chowdhury, H.-M. Tsou, Z. M. Jiang, and A. E. Hassan, "Understanding the rationale for updating a functions comment," in *Software Maintenance, 2008. ICSM 2008. IEEE International Conference on*. IEEE, 2008, pp. 167–176.

[16] G. Sridhara, "Automatically detecting the up-to-date status of todo comments in java programs," in *Proceedings of the 9th India Software Engineering Conference*. ACM, 2016, pp. 16–25.

[17] H. C. Gall, B. Fluri, and M. Pinzger, "Change analysis with evolizer and changedistiller," *IEEE Software*, vol. 26, no. 1, p. 26, 2009.

[18] T. Mens and T. Tourwé, "A survey of software refactoring," *IEEE Transactions on software engineering*, vol. 30, no. 2, pp. 126–139, 2004.

[19] S. Margaret-Anne, J. Ryall, R. I. Bull, D. Myers, and J. Singer, "Todo or to bug: Exploring how task annotations play a role in the work practices of software developers," in *Proceedings of the 30 th Int? l. Conf. Software Engineering(ICSE? 08)*, pp. 251–260.

[20] X. Ye, H. Shen, X. Ma, R. Bunescu, and C. Liu, "From word embeddings to document similarities for improved information retrieval in software engineering," in *Ieee/acm International Conference on Software Engineering*, 2016, pp. 404–415.

[21] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," *Advances in Neural Information Processing Systems*, vol. 26, pp. 3111–3119, 2013.

[22] A. Liaw and M. Wiener, "Classification and regression by randomforest," *R News*, vol. 23, no. 23, 2002.

[23] H. He and E. A. Garcia, "Learning from imbalanced data," *IEEE Transactions on knowledge and data engineering*, vol. 21, no. 9, pp. 1263–1284, 2009.