A Doc2Vec-Based Assessment of Comments and Its Application to Change-Prone Method Analysis

Hirohisa Aman*, Sousuke Amasaki†, Tomoyuki Yokogawa† and Minoru Kawahara*
*Center for Information Technology, Ehime University, Matsuyama, Ehime, 790–8577 Japan
†Faculty of Computer Sc. & Systems Eng., Okayama Prefectural University, Soja, Okayama, 719–1197 Japan

Abstract—**Comments in a source program can be helpful artifacts for program comprehension. While many comments are useful documents embedded in source programs, there are also poorly-informative comments in the real world. In order to quantitatively assess the value of comments, this paper proposes applying the Doc2Vec model to comment evaluation. Doc2Vec is a useful model for vectorizing the content of a document. In this paper, a Java method is regarded as a document, and its content is expressed as a vector. Then, two vectors corresponding to different versions of a method are prepared—the original version and the comment-erased version—, and the vector similarity between these two versions are computed. If the erased comments provided richer information for the source code, the corresponding vector would have a larger change through the comment elimination. A method having poorly-informative comments may be low-quality and might require more code modifications. This paper analyzes the relationship between the value of comments in a method and the change-proneness, using the data collected from five popular open source software projects. The results show that a method having poorly-informative comments is likely to be change-prone, i.e., such a method could not survive unscathed after release.**

I. Introduction

Programmers may write comments in their source programs. While comments have no impact on the program behavior, they may provide additional information about the program and can be useful in program comprehension [1]. For instance, when a programmer writes a method (function), he/she often includes comments on how to use the method as well. Such comments can be a useful manual and a help during the program's reuse. In an informal fashion, programmers may also add comments to their code fragments within a method body, e.g. as memos explaining their decisions in the source code. However, not all comments are always useful. Martin [2] provided 18 "bad" comment categories such as "redundant" comments, "noise" ones and so on. Corazza et al. [3] reported that obsolete comments have harmful effects on the code quality. Kernighan and Pike [4] suggested not adding comments to "bad" code, and recommended rewriting the code itself rather than adding descriptive comments. Similarly, Boswell and Foucher [5] stated that good code is clear and does not require any additional comments to understand it.

While there have been studies focusing on the presence of comments or the amount of them in the past (e.g., [6]–[8]), machine learning-based approaches to comments have also become more and more popular in recent years (e.g., [9]–[11]). Pascarella and Bacchelli [11] analyzed the content of comments, and performed a comment categorization using a machine learning method. They reported that 59% of comment lines are not directly related to the commented source code, and emphasized the importance of understanding the content of comments from a semantic perspective. Thus, in this paper, we propose quantitatively assessing the value of comments by leveraging a well-designed natural language processing (NLP) model, Doc2Vec [12], [13]. In our proposal, we regard a method (function) as a document, and express the method's content as a vector by using the Doc2Vec model. After that, we erase comments from the method and express it as a vector again. Then, we compare the vectors before and after the comment elimination. As the erased comments provide richer information, the difference between vectors gets larger. This is the key of our idea of comment assessment in this paper.

The key contributions of this paper are as follows:

1) We proposed a novel method for assessing the value of comments in Java methods, which evaluates how rich the information provided by the comments is.
2) We collected empirical data from five popular open source software (OSS) projects, and examined the effect of our method through a change-prone method analysis. This is because a method having poorly-informative comments may be low-quality and might require more code changes after its release. The results showed that a method having poorly-informative comments is likely to be change-prone, i.e., such a method would require more code changes after the release. Our dataset is available at https://bit.ly/2Pb89Au .

The remainder of this paper is organized as follows: Section II briefly explains the comments of interest in this paper, and Section III describes the proposed method using Doc2Vec. Section IV presents a data analysis as an application of the proposed method, and Section V describes the related work. Finally, Section VI gives our conclusion and future work.

II. Comments of Interest

Comments can be classified into various categories in accordance with their aims and positions [10]. Let us take a simple example shown in Fig. 1, which is a part of the Java source file. In the figure, four different types of comments appear: (A) a copyright designation, (B) a simple description of the class, (C) a programmer's manual for the method, and (D) an explanation of the code fragment. Comment (A) is just for presenting the copyright and does not give any explanation about the implementation. Comments (B) and (C) play roles as manuals explaining the aim of the corresponding class and how to use the corresponding method, respectively. Comment (D) presents an explanation of the reason why the following code performs such an operation. Needless to say, there are other types of comments as well; we omit them due to space limitations, see [10] for more details.

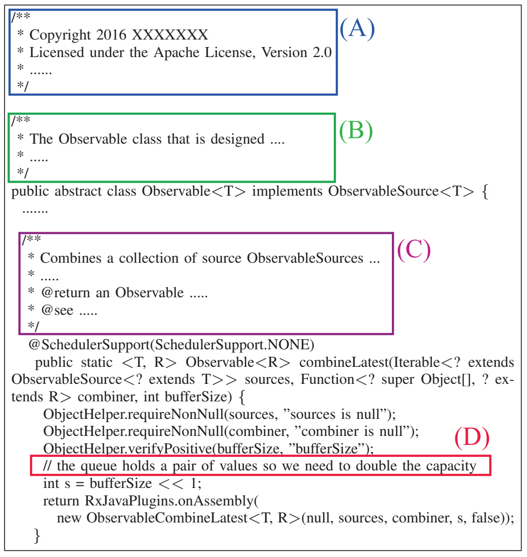
In Java programs, comments like (B) and (C) are also used for the automatic generation of a manual through the Javadoc. On the other hand, comments written inside a method body like (D) are not used for the manual generation, but they are usually referred to when programmers review the code. Hence, type (D) comments would play an important role in code comprehension: the author of the code might be the only person who can quickly and properly understand the reason why the following code "bufferSize << 1" was included without the comment. We will focus on such comments, i.e., comments written inside a method body, and propose quantitatively assessing the value of comment in a method.

III. Evaluation of Comment Impact Using Doc2Vec

A. Doc2Vec

In order to assess comments, we leverage Doc2Vec, a well-designed NLP technology, which utilizes a machine learning method with a neural network.

A Doc2Vec model can express a document as a vector. We can evaluate "semantic" similarity between two documents by comparing the corresponding vectors. Doc2Vec is based on Word2Vec [14] which expresses a word as a vector. In a vector space produced by a Word2Vec model, two words which are similar in meaning correspond to two vectors which are close to each other. Furthermore, the relationship among words is consistent throughout vector operations, e.g., "king – man + woman = queen." Doc2Vec is extended to achieve such a successful vectorization for an entire document. Its algorithm is implemented in the gensim [13], a Python library.



```
/**
 * Copyright 2016 XXXXXXX
 * Licensed under the Apache License, Version 2.0
 * ......
 */

/**
 * The Observable class that is designed ....
 * .....
 */
public abstract class Observable<T> implements ObservableSource<T> {
  .......

  /**
   * Combines a collection of source ObservableSources ...
   * .....
   * @return an Observable .....
   * @see .....
   */
  @SchedulerSupport(SchedulerSupport.NONE)
   public static <T, R> Observable<R> combineLatest(Iterable<? extends
ObservableSource<? extends T>> sources, Function<? super Object[], ? ex-
tends R> combiner, int bufferSize) {
     ObjectHelper.requireNonNull(sources, ”sources is null”);
     ObjectHelper.requireNonNull(combiner, ”combiner is null”);
     ObjectHelper.verifyPositive(bufferSize, ”bufferSize”);
     // the queue holds a pair of values so we need to double the capacity
     int s = bufferSize << 1;
     return RxJavaPlugins.onAssembly(
        new ObservableCombineLatest<T, R>(null, sources, combiner, s, false));
   }
```

Fig. 1. Example of a commented program.

B. Impact of Comments on Source code

Now we regard a method (function) as a document, and consider applying Doc2Vec to express the method content as a vector. We prepare two versions of a method, the original version and the comment-erased version, then we compute the cosine similarity between two vectors corresponding to these versions (see Fig. 2). If a comment provided useful and rich information in the original code, the corresponding vector would vary significantly from the comment-erased vector. That is to say, two vectors corresponding to the original version and the comment-erased version would have a relatively low level of similarity. On the other hand, if a comment gives not-so-rich information (see Fig. 3 for example), the corresponding vector would have a small change after the comment elimination, and the vector of the original code would have a high similarity to the vector of the comment-erased one.

Notice that "rich" comments do not always mean "careful" ones. Even carefully-written comments could be poorly-informative when the comments do not provide any additional information. For example, when we have a method invocation "sort(data);" in a program, a comment "// sort the data" is careful but poor, i.e., it gives no additional information.

C. Comment Assessment Method

We explain the concrete steps of our proposal in this subsection. Since our focus is on the comments written inside a method (function) body, we regard a method as a document in the analysis using Doc2Vec.

1) Doc2Vec model construction:
   We need to build a Doc2Vec model trained for the target software. To train a model, we have to prepare a corpus of tokens appearing in source programs.
   At first, we collect source files from the target software, and extract the methods from them through a syntax analysis. After that, we tokenize the content of each method, and make each method's token list in which

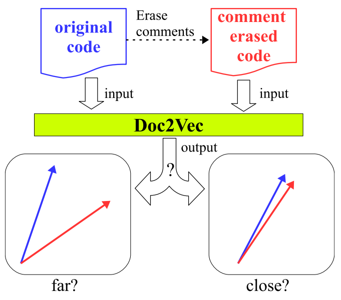
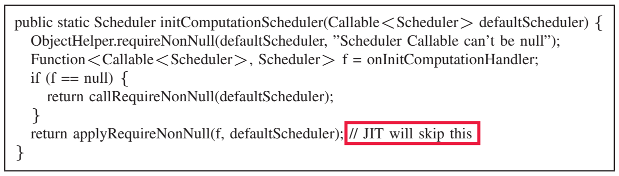


Fig. 2. An image structure of the proposal.



```
public static Scheduler initComputationScheduler(Callable<Scheduler> defaultScheduler) {
   ObjectHelper.requireNonNull(defaultScheduler, ”Scheduler Callable can’t be null”);
   Function<Callable<Scheduler>, Scheduler> f = onInitComputationHandler;
   if (f == null) {
      return callRequireNonNull(defaultScheduler);
   }
   return applyRequireNonNull(f, defaultScheduler); // JIT will skip this
}
```

Fig. 3. Another example of a method having comments.

the programming language's reserved keywords, symbols (including operators) and English stop words[1] are excluded. For example, we obtain the token list shown in Fig. 4 for method "combineLatest" appearing in Fig. 1. By gathering such a token list, we prepare a corpus in which each line corresponds to each method's token list. Then, build a Doc2Vec model by using the corpus as a training dataset.

2) Preparation of token list pair corresponding to the original version and the comment-erased version:
For a target method m, we extract m's token list from the corpus—we call it $T_0(m)$. Next we extract m's content from the source file and erase its all comments. Then, we make m's token list again—we refer to it as $T_1(m)$.
3) Similarity computation:
By using the model constructed in step 1), we compute the cosine similarity between $T_0(m)$ and $T_1(m)$—we call it $S_C(m)$. The lower $S_C(m)$ value is, the more changed m's semantic vector was, through the comment elimination. So, the erased comment would provide richer information to the source code.

□

As an example, we computed $S_C(\cdot)$ for the method shown in Fig. 1: the similarity between the original version and the comment-erased one was 0.568. On the other hand, the similarity for the method shown in Fig. 3 was 0.979. That is to say, we successfully evaluated that the comments within the method shown in Fig. 1 seem to provide richer information than the one shown in Fig. 3.

IV. Data Analysis

A. Aim and Dataset

In this section, we report the data analysis that we conducted and how it can be used in quality management. In general, comments tend to enhance the understandability of a program. However, if a comment provides poor information, it would not be a help to programmers. Although it is the best to examine our proposal through a manual experiment of comment evaluation involving many programmers, that is costly and not easy to perform. Therefore, we adopted an indirect measure: the change-proneness of a method, which is quantified by the number of code changes that have occurred in the method. We conducted a data analysis to examine the relationship between our comment evaluation and the change-proneness of methods.

Our dataset consists of methods collected from five OSS projects shown in Table I. We selected these projects for the following two reasons: 1) their source files are written in Java, and 2) they are popular projects. The first reason is due to our data collection tool since we need to do a syntax analysis and we developed a tool to analyze Java methods for this study. The second reason is for enhancing the worth of the analysis results. Results derived from more popular projects would be more attractive for more developers and researchers. These are popular projects which ranked in the top 10 in terms of GitHub "stars" at the time that we conducted this study[2].

TABLE I
Analyzed OSS projects.

project	analyzed version	size (KLOC)	# of methods	# of commented methods
Elasticsearch	5.0.0α1	546	12,354	220
Guava	7.0	49	5,959	294
OkHttp	3.0.0	46	820	57
RxJava	2.0.0	214	2,986	290
Spring Boot	1.0.0	51	2,748	66
total		906	24,867	927

B. Procedure

We conducted our data analysis for each project using the following procedure. The collected data and the constructed Doc2Vec models are available at https://bit.ly/2Pb89Au .

1) We built a Doc2Vec model by using all methods included in the project.
2) For each commented method m, we computed the evaluation of its comments as $S_C(m)$.
3) For each commented method m, we counted the occurrences of its code change events (commits) after the release of the analyzed version; the analyzed version is shown in Table I.
4) We categorized the commented methods into four categories according to the quartile of $S_C(m)$ distribution—C1: $S_C(m) < Q_1$; C2: $Q_1 \leq S_C(m) < Q_2$; C3: $Q_2 \leq S_C(m) < Q_3$; C4: $S_C(m) \geq Q_3$, where Q_1, Q_2 and Q_3 are the 1st, 2nd and 3rd quartile, respectively. Then, we compared the number of code change events[3] that occurred in the methods among these categories.

C. Results

Table II and Fig. 5 show comparisons of method categories in terms of the change-proneness: they present the mean and the standard error of the change counts within each category of each project, respectively; In Table II, the highest mean value within a project is emphasized by the boldface and underline.

The highest mean number of change counts within each software project is observed at C4 or C3. Although not all projects showed monotonic increase trends of the mean of change counts from C1 to C4, they seem to have a tendency where the methods in categories C3 and C4 are likely to be more change-prone than the ones in C1 and C2. That is to say, a method having a higher $S_C(m)$—the one whose comments provided poorer information—would be more change-prone.

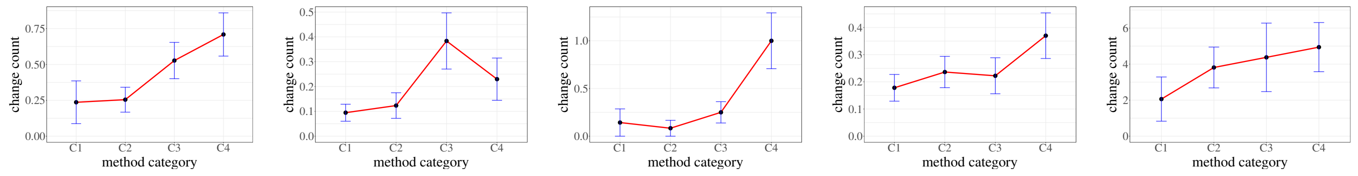
```
T R Observable R combineLatest Iterable ObservableSource T sources Function
Object R combiner bufferSize ObjectHelper requireNonNull sources sources
ObjectHelper requireNonNull combiner combiner ObjectHelper verifyPositive
bufferSize bufferSize queue holds pair values need capacity s bufferSize 1
RxJavaPlugins onAssembly ObservableCombineLatest T R sources combiner s
```

Fig. 4. Token list of method "combineLatest" shown in Fig. 1.

[1]https://www.textfixer.com/tutorials/common-english-words.txt

[2]We checked it on March 10th, 2018.

[3]These change events were counted by using the commit logs stored in the repositories which were cloned on March 10th, 2018.



(a) Elasticsearch (b) Guava (c) OkHttp (d) RxJava (e) Spring Boot

Fig. 5. Mean change count and standard error for each category of each project.

TABLE II
MEAN CHANGE COUNT ($\overline{x}$) AND STANDARD ERROR (SE) FOR EACH CATEGORY OF EACH PROJECT.

project		method category			
		C1	C2	C3	C4
(a) Elasticsearch	$\overline{x}$	0.236	0.255	0.527	**0.709**
	SE	0.149	0.087	0.127	0.151
(b) Guava	$\overline{x}$	0.095	0.123	**0.384**	0.230
	SE	0.034	0.051	0.113	0.085
(c) OkHttp	$\overline{x}$	0.143	0.083	0.250	**1.000**
	SE	0.143	0.083	0.112	0.293
(d) RxJava	$\overline{x}$	0.178	0.236	0.222	**0.370**
	SE	0.049	0.058	0.066	0.084
(e) Spring Boot	$\overline{x}$	2.059	3.813	4.375	**4.941**
	SE	1.226	1.134	1.901	1.365

D. Discussions

As a result of our data analysis, we observed that a method, whose vector did not significantly change through the comment elimination, tends to be change-prone. We cannot say it is a strong trend due to the observed standard error (SE). Nonetheless, the difference of mean change counts between C1 and C4 are clear for all projects (see Fig. 5), so it seems to be reasonable to say that a method having a high $S_C(\cdot)$ value tends to be change-prone. In such a method, the erased comments would have smaller impacts on the code, i.e., the comments would be poorly-informative. Hence, such comments would not be a help to programmers, and they might be related to a lower degree of perfection of the method.

In the above results, project "(b) Guava" showed a different trend in which the maximum mean change count was observed in C3 rather than C4. By performing additional research on the $S_C(\cdot)$ value distribution, we found that the Guava project had a different tendency involving $S_C(\cdot)$ values: its average was 0.720 which was the lowest value among the projects (the mean was 0.823 and the standard deviation was 0.072). That is to say, Guava is more likely to have more large-impact comments than other projects. Such a difference may cause the slightly different trend. Nonetheless, since the change rates in C1 and C2 are less than that in C3 and C4, a method whose comments provide richer information tends to be less change-prone, and this trend is the same as the others. Hence, the poorness of comments evaluated by our Doc2Vec-based method would be related to the change-proneness of Java methods.

However, we have a concern that the length of the method might also affect the above results. We adopted Doc2Vec for evaluating comments in Java methods since the size (the number of dimensions) of vectors generated by the same Doc2Vec model is constant regardless of the length of documents, i.e., Java methods. In this way, we can easily compare two different versions of methods before and after the comment elimination even though these methods have different lengths. Nonetheless, the vectors before and after the comment elimination may have a higher similarity if the method is longer and has a lower proportion of the corresponding comments. For each project, we computed the Spearman's rank correlation coefficient (ρ) between the method length and $S_c(\cdot)$. Then, two out of five projects showed strong correlations ((a) Elasticsearch: $\rho = 0.701$; (c) OkHttp: $\rho = 0.729$)[4]. For these two projects, the length of the methods seem to be a confounding factor and we cannot properly assess the power of our proposed model. Thus, we need to examine whether the comment evaluations in such Java methods were appropriate or not, so we would like to perform manual evaluations involving many programmers and to do a further examination in the future. Moreover, it might also be effective to focus only on a narrower range around the comments rather than the whole Java method. This is also a significant part of our future work.

E. Threats to Validity

Since a Doc2Vec model may produce a different vector for the same content if it was trained by a different corpus, the corpus selection is our major threat to validity. To mitigate this threat, we used a major and not-early release version of a target software because a non-major version or an early version might be at an immature state. In order to evaluate the influence of the selected corpus, we would like to conduct a sensitivity analysis in the future.

While we regarded an identifier as a word in our study, it might be noise in analyzing the semantics. For example, it might be better to split "bufferSize" into two words "buffer" and "size." We plan to perform a further analysis leveraging an identifier splitting technique [15] as our future work.

Although the Doc2Vec model is a successful model for vectoring the content of documents and our proposal is based on the assumption that a comment would be written for giving information related to the source code, there is also room for a further study in order to accurately understand the value of comments. For example, even if there are long comments which are totally irrelevant to the source code, our Doc2Vec-based method would wrongly rate such comments as

[4]The remaining projects did not have strong correlations: $\rho = 0.423$, 0.395 and 0.078 for (b) Guava, (d) RxJava and (e) Spring Boot, respectively.

informative ones since they seem to add additional information with many (but irrelevant) words. Our above assumption can be a threat to validity, and we need to perform a further study together with manual evaluations of comments in the future.

V. Related Work

Thomas et al. [16] leveraged the topic model to evaluate a semantic similarity among source programs, and applied the computed similarities to prioritize test cases (programs) for an efficient regression testing. In their approach, they tried obtaining a higher test coverage by selecting fewer test cases which are dissimilar to the already-selected test cases. Their study is a notable previous work applying an NLP technique to source program analysis. While the fundamental notion—an application of an NLP technique to evaluate a similarity between programs—is common, we utilized such a technique for evaluating the comments.

Steidl et al. [10] analyzed a correspondence relationship between the content of comments and the commented code. They proposed evaluating those comments by using the occurrence rate of words which are common or similar to the corresponding identifiers. For example, when there is a comment "removes all defined markers" followed by method "removeAllMarkers()," three out of four words in the comment appear in the method's name, i.e., the occurrence rate is 0.75. Steidl et al. considered a comment would not provide useful information if its occurrence rate is greater than 0.5. Their work is a remarkable previous work to evaluate the usefulness of comments. We focused on yet another point of view related to the semantics through the Doc2Vec model.

Corazza et al. [3] focused on the "coherence" of comments and corresponding implementations of methods—a coherent method has well-written lead comments which properly describe the goal and/or the implementation details of the methods, together with the details about the parameters and their intended use. Through a lot of manual assessments on Java methods collected from open source applications, they created a useful dataset of method coherence, and discussed the real trends of relationships between method implementations and their lead comments. While our fundamental concept of comment assessment is common to their work, we tried making assessments of comments from a perspective of document similarity. By performing a manual assessment similar to the work of Corazza et al., we would prepare a better corpus for training our Doc2Vec model. It is another significant part of our future work.

VI. Conclusion

We focused on the comments written within a method body, and proposed evaluating these comments by using a Doc2Vec model, a popular NLP model. In our proposal, for a method having comments, we prepare two versions of the method: the original version and the comment-erased version. Then, we compare these two versions using the Doc2Vec vectorization and the computation of cosine similarity between the corresponding vectors. If a comment provided richer information in a method, the corresponding vector would change significantly due to the comment elimination: this is our key idea. As an application of our proposal, we conducted a data analysis using five popular OSS projects. Then, we found that a method whose comments provided poorer information tended to be more change-prone. Poorly-informative comments would not be a help to programmers, and they might be related to a lower degree of perfection in the method.

Our future work includes: (1) a further examination of the proposed model by using manual evaluations; (2) further analyses with different corpora produced by using source files of earlier or later versions, or by performing a more detailed analysis of identifiers; (3) an improvement of our model to reduce the impact of the method length, e.g., by focusing on a narrower range around the comments.

Acknowledgment

This work was supported by JSPS KAKENHI #16K00099 and #18K11246. The authors would like to thank the reviewers for their helpful comments on an earlier version of this paper.

References

[1] S. C. B. de Souza, N. Anquetil, and K. M. de Oliveira, "A study of the documentation essential to software maintenance," in *Proc. 23rd Int'l Conf. Design of Communication*, Sept. 2005, pp. 68–75.

[2] R. C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship*. Boston, MA: Prentice Hall, 2008.

[3] A. Corazza, V. Maggio, and G. Scanniello, "Coherence of comments and method implementations: a dataset and an empirical investigation," *Softw. Quality J.*, pp. 1–27, Nov. 2016.

[4] B. W. Kernighan and R. Pike, *The practice of programming*. Boston, MA: Addison-Wesley Longman, 1999.

[5] D. Boswell and T. Foucher, *The Art of Readable Code: Simple and Practical Techniques for Writing Better Code*. Sebastopol, CA: Oreilly & Associates, 2011.

[6] P. W. Oman and J. Hagemeister, "Metrics for assessing software system maintainability," in *Proc. Conf. Softw. Maintenance*, Nov. 1992, pp. 337–344.

[7] M. J. B. García and J. C. Granja-Alvarez, "Maintainability as a key factor in maintenance productivity: a case study," in *Proc. Int'l Conf. Softw. Maintenance*, Nov. 1996, pp. 87–93.

[8] H. Aman, S. Amasaki, T. Sasaki, and M. Kawahara, "Lines of comments as a noteworthy metric for analyzing fault-proneness in methods," *IEICE Trans. Inf. & Syst.*, vol. E98-D, no. 12, pp. 2218–2228, Dec. 2015.

[9] L. Tan, D. Yuan, G. Krishna, and Y. Zhou, "/*icomment: bugs or bad comments?*/," in *Proc. 21th ACM SIGOPS Symp. Operating Syst. Principles*, Oct. 2007, pp. 145–158.

[10] D. Steidl, B. Hummel, and E. Juergens, "Quality analysis of source code comments," in *Proc. 21st Int'l Conf. Program Comprehension* , May 2013, pp. 83–92.

[11] L. Pascarella and A. Bacchelli, "Classifying code comments in java open-source software systems," in *14th Int'l Conf. Mining Softw. Repositories*, May 2017, pp. 227–237.

[12] J. H. Lau and T. Baldwin, "An empirical evaluation of doc2vec with practical insights into document embedding generation," in *Proc. 1st Workshop Representation Learning for NLP*, Aug. 2016, pp. 78–86.

[13] R. Řehůřek, "gensim: models.word2vec—deep learning with word2vec," https://radimrehurek.com/gensim/models/word2vec.html, Mar. 2018.

[14] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *CoRR*, vol. abs/1301.3781, Sept. 2013.

[15] D. Binkley, D. Lawrie, L. Pollock, E. Hill, and K. Vijay-Shanker, "A dataset for evaluating identifier splitters," in *Proc. 10th Working Conf. Mining Softw. Repositories*, May 2013, pp. 401–404.

[16] S. W. Thomas, H. Hemmati, A. E. Hassan, and D. Blostein, "Static test case prioritization using topic models," *Empir. Softw. Eng.*, vol. 19, no. 1, pp. 182–212, Feb. 2014.