SOFTWARE ENGINEERING, SOFTWARE TESTING AND VERIFICATION

CALCULATING SIMILARITY OF JAVADOC COMMENTS

© **20XX y.** D.V. Koznov[1], E.Yu. Ledeneva[2], D.V. Luciv[1], P.I. Braslavski[3]
[1] *Saint Petersburg State University, 199034 St. Petersburg, Universitetskaya Emb., 7–9*
[2] *Yandex LLC, 119021 Moscow, Lva Tolstogo st., 7*
[3] *HSE University, 109028 Moscow, Pokrovsky blvd., 11*
E-mail: d.koznov@spbu.ru; ekiuled@gmail.com; d.lutsiv@spbu.ru; pbras@yandex.ru

Code comments are an essential part of software documentation. Many software projects suffer from the problem of low-quality comments that are often produced by copy-paste. In case of similar methods, classes, etc. copy-pasted comments with minor modifications are justified. However, in many cases this approach leads to degraded documentation quality and, subsequently, to problematic maintenance and development of the project. In this study, we address the problem of near-duplicate code comments detection, which can potentially improve software documentation. We have conducted a thorough evaluation of traditional string similarity metrics and modern machine learning methods. In our experiment, we use a collection of Javadoc comments from four industrial open-source Java projects. We have found out that LCS (Longest Common Subsequence) is the best similarity algorithm taking into account both quality (Precision 94%, Recall 74%) and performance.

Keywords: software documentation, Javadoc comments, similarity measure

1. INTRODUCTION

Code documentation is one of the most important types of software documentation. It is essential for overall software maintenance and especially for application-program interfaces (APIs), which are intended to be used by other developers [1].

Special documentation tools (e.g., Javadoc, Doxygen) produce readable and navigable web-based code documentation from tagged code comments and other artifacts (e.g., standalone texts, .dot format diagrams, method signatures, etc.). Javadoc is a de-facto standard for Java applications. It includes a markup language and a tool, which are integrated into most Java IDEs such as Eclipse, IDEA, etc.

There are many duplicates in comments because copy-paste is often used to write a new comment for a new class, method, etc. [2, 3]. This is widely practised because very often new code entities are significantly functionally similar to a previously created one. Consequently, their corresponding comments are very similar as well. The copy-pasted comment is modified in accordance to the specifics of the code entity it belongs to, and thus spawns not an exact duplicate, but a near duplicate instead. However, it is possible for copy-pasted comments to contain incorrect information due to various reasons. One of them is the developers' lack of time for less prioritized tasks: a developer might copy-paste an old comment, change it slightly, postpone a final revision, and, finally, forget about it altogether. Errors in changes are also possible (e.g. forgetting to change some names in a copy-pasted comment). Thus, a presence of near duplicates is considered normal practice, but due to the aforementioned problems, there is a need for specialized tools for their detection and analysis to identify inconsistencies between comments and code [2].

A number of papers has been dedicated to duplicates in code comments [2, 3, 4, 5]. Nevertheless, not enough attention is paid to both automatic detection of duplicates and search of near duplicates. As a matter of fact, only [3] suggest a toolset for duplicate detection, but it does not deal with near duplicates.

This paper closes the gap considering near

duplicates in Javadoc comments. We explore various state-of-the-art algorithms to evaluate pairwise similarity of Javadoc comments, considering both string matching algorithms and modern machine learning (ML) algorithms. For our experiments, we have used four widely known open source Java projects: GSON, JUnit4, Mockito, and Slf4J. We have found out that LCS (Longest Common Subsequence) is the best similarity algorithm taking into account both quality (Precision 94%, Recall 74%) and performance.We have found a reasonable number of near duplicate pairs and detected some copy-paste errors.

2. BACKGROUND

2.1. NEAR DUPLICATE JAVADOC COMMENTS

Let us consider two methods from the JUnit4 project: `assertTrue` and `assertFalse`. The first one throws an exception if its argument (a condition) is true, the next one does the same when its argument is false. The functionality of these methods is very similar, and consequently, their comments should also be very alike. Indeed, these comments differ by one word only: it is **bold** in the example below. Thus, they are near duplicates.

```
/** Asserts that a condition is true. If it
    isn't it throws an AssertionError
    without a message.
    @param condition condition to be checked */
public static void assertTrue(boolean condition)

/** Asserts that a condition is false. If it
    isn't it throws an AssertionError
    without a message.
    @param condition condition to be checked */
public static void assertFalse(boolean condition)
```

2.2. SELECTED SIMILARITY ALGORITHMS

A similarity function is necessary to detect near duplicate comments (strings). We consider the most well-known state-of-the-art similarity algorithms, both string matching and machine learning: Longest Common Subsequence (LCS), Cosine Similarity (COS), Locality-Sensitive Hashing (LSH), Word Mover's Distance (WMD), doc2vec (D2V), and Siamese Neural Network (SNN). A short overview of the algorithms is presented in Table 1.

Table 1.: A Brief Overview of Algorithms Selected

Algorithm	Description
LCS [6]	Measures the total length of the longest matching substrings in both strings (text fragments) appearing in the same order.
COS [7]	Transforms a string into term frequency (TF) into multidimensional vector. Each component of the vector is proportional to the frequency of use of this word in the text and inversely proportional to the frequency of use of the word in the totality of the analyzed texts (inverse document frequency, IDF). The similarity of these vectors is calculated as the cosine of the angle between them.
LSH [8]	Represents a string as a bag of n-grams (series of several adjacent words), to which the minhash algorithm [9] is applied, splits the resulting bit vectors into fragments and calculates the proportion of matching vector fragments resulting from different strings.
WMD [10]	Represents two strings as clouds of points in a vector space (embeddings), and calculates their similarity as the cost of moving one cloud into the other. It uses word2vec to construct embeddings for words of the string, based on a neural network model that learns word associations from a large corpus of text.
D2V [11]	An extension of word2vec that constructs embeddings for entire documents rather than individual words.
SNN [12]	A neural network with two subnetworks sharing both architecture and weights. Our implementation uses LSTM (Long Short Term Memory) components.

LCS, COS, and LSH are widely known string matching algorithms which are based on different ideas. We have used implementations of LCS from the difflib library[1], COS from the scipy library[2] and the LSH implementation was adopted from the datasketch library[3].

WMD, D2V, and SNN are some of the leading and most promising machine learning algorithms for text processing. We have adopted WDM and D2V from gensim[4] and SNN from Keras[5].

3. RELATED WORK

Very often code comments are not of very high quality — they may be incomplete and contain errors and inconsistencies [13, 14]. Moreover, if the system is big enough, often its code comments do not have a unified style, because different developers write in different manners [15]. There is a number of research methods and tools for detection of errors and inconsistencies in code comments [5, 16, 17, 18, 19, 20]. But the task is still far from being solved, so it is open for new approaches.

There is some research on duplicates in software documentation — a survey can be found in [15, 21]. In [15, 21, 22, 23] near duplicates in software documentation are considered, but these papers do not deal with Javadoc comments. Nasal and Poruban [4] consider near duplicates in Javadoc. However, they only present a Javadoc extension for comment reuse, and no tools for detection of near duplicates. Oumaziz et al. [2] and Blasi et al. [3] have explored the connection between duplicate Javadoc comments and duplicates in code, but they did not consider near duplicates. Wagner and Fernández [24] wrote about similarity algorithms for software texts, but they have given only an overview of state-of-art approaches without considering Javadoc. Meanwhile, Javadoc is a popular approach for commenting Java code and is used in most Java projects.

Existing research on Javadoc duplicates is that it does not use modern similarity algorithms for duplicate detection. Additionally, it does not take into account near duplicates properly.

4. EXPERIMENT DESIGN

In [15, 21], duplicate fragments are considered without taking into account the structure of the document, which leads to the detection of a large quantity of meaningless duplicates. Furthermore, the capability to search for duplicated Javadoc tags is offered [2]. However, this approach finds too many duplicates, which are additionally very fine-grained, and the obtained information is too complex to analyze. We have chosen to search for near duplicates in whole comments that belong to specific code entities (such as a class, method, annotation, etc.). This allows us to operate with meaningful text fragments.

We have used well-known Java projects for our experiments, presented in Table 2. Besides the volume of source code, we have also reported the number of classes, methods, and comments for every project, as well as the number of duplicate pairs we found in these projects for our experiments.

We have involved two industrial Java developers as experts in our study. To mitigate threats to validity, they both worked independently on the same data, after which they have discussed their results and resolved conflicting decisions.

We have created our expert-annotated dataset in the following way. First, we have connected all source text of all projects in a single file. Next, we have kept only Javadoc comments and their corresponding code entities in the file. After that, we have applied Clone Miner (a token-based software clone detection tool) [25] and obtained a set of clone groups in comments, with minimum size of 5 words in a clone, following [21]. For each clone group G, we split G into two groups if a clone of G contained text from two neighboring comments (Clone Miner does not consider the structure of the analyzed text): in our study, a found clone did not contain text from more than two neighboring comments. Next, we have sorted these groups by clone size. Afterwards, we have replaced each group G with G': for each text fragment $g \in G$ the comment g' that contains the fragment g was recovered and added to G'. Further on, the experts have manually analyzed each comment group and

[1] https://docs.python.org/3/library/difflib.html
[2] https://scipy.org
[3] http://ekzhu.com/datasketch/
[4] https://radimrehurek.com/gensim/
[5] https://keras.io/

Table 2.: Evaluated projects

Name	Size, KB	Classes	Methods	Comments	Pairs
GSON[6]	480	50	281	426	364
JUnit4[7]	588	103	433	602	744
Mockito[8]	901	110	471	669	786
Slf4J[9]	163	21	158	227	706

selected pairs of similar comments. We have used Duplicate Finder [23] to visualize duplicate fragments of comments in a group. Therefore, pairs marked as positive for similarity were constructed from the comments from within a single clone group. To the obtained set of positives (similar pairs), we have automatically added the same number of negatives (dissimilar pairs) from different groups, which were then additionally reviewed by the experts. The whole process was facilitated with a number of Python scripts developed by us.

As the result, our expert-annotated dataset consists of 2600 comment pairs in total (see the "Pairs" column in Table 2). The average length of a Javadoc comment in the considered projects is 52 words.

When deciding whether a pair is positive or negative, the experts considered not only the textual similarity of comments, but also the likeness of functionality that these comments describe. There are situations in which the comments are similar textually, but this similarity is provided only by standalone words or short word combinations. These pairs were marked as negative. Another situation is comments in a pair having a single common tag (e.g., the same exception), but the functionality of the corresponding program entities is very different. These pairs were also marked as negative. At last, it is possible for short comments to include the same word combination describing the same program object (e.g. "tests to be run by the receiver"), but the whole respective functionality to be completely different. In this case, the experts also marked the pair as negative.

We have conducted two experiments. Experiment 1 was performed on the expert-annotated dataset with a 70/30 train-test split. We have calculated a threshold for every algorithm maximizing F1 on the train part of the dataset. Additionally, to adapt the machine learning algorithms (WMD, D2V, SNN) to our task, we trained their embeddings on a large dataset of Javadoc comments found on Kaggle[10] (around 2 000 000 comments).

To obtain more realistic results, we have conducted Experiment 2 substituting test data with a full set of Javadoc comments of the JUnit4 project. Here we excluded JUnit4 pairs, used as a part of training data before. Since new test data were not fully annotated by experts (when creating the expert-annotated dataset, we did not detect all of positive comment pairs in considered projects), they verified additional positive pairs from the entire output of Experiment 2 manually. To simplify their work, they have used the Duplicate Finder visualizer [23] to highlight the similar text in the pairs.

Since our input data for both experiments was not balanced regarding to the number of positive and negative pairs, we drew conclusions about the results of our experiments based on three metrics — accuracy (P, Precision), completeness (R, Recall) and measure F1.

5. RESULTS AND DISCUSSION

The results of our study are presented in Table 3. Columns 2–4 represent Experiment 1, and columns 5–8 contain the results of Experiment 2. In Experiment 1 F1 oscillates from 0.94 to 0.97. String matching algorithms show stable Precision (0.97), whereas for ML algorithms Precision ranges from 0.92 (SNN) to 0.98 (WMD).

The results we obtained in Experiment 2 are worse than in Experiment 1. This can be explained by the fact that in Experiment 1 we have used a dataset created manually by experts.

[6] https://github.com/google/gson
[7] https://github.com/junit-team/junit4
[8] https://github.com/mockito/mockito
[9] https://github.com/qos-ch/slf4j
[10] Kaggle dataset, https://www.kaggle.com/isofew/code-comment-pairs

Table 3.: Experiment results

Algs	Experiment 1			Experiment 2			
	Prec	Rec	F1	Prec	Rec	F1	Time, s
LCS	0.97	0.95	0.96	0.96	0.74	0.84	15
COS	0.97	0.94	0.96	0.88	0.82	0.85	19
LSH	0.97	0.90	0.94	0.93	0.69	0.80	625
D2V	0.94	0.99	0.97	0.64	0.90	0.75	3157
WMD	0.98	0.95	0.96	0.97	0.73	0.84	1949
SNN	0.92	0.97	0.95	0.79	0.86	0.82	82

Consequently, the training data consisted of well-separated pairs, i.e. their scores were close to 1 for positives and to 0 for negatives. When carrying out Experiment 2, we have had a larger number of pairs having similarities close to the threshold value separating negatives and positives. They should have been used as training data, but due to the high complexity of expert's estimation (a reasonable number such pairs were found) , this was not done. Machine learning algorithms used in our research, especially D2V, turned out to be more sensitive to this specific of input data than string algorithms.

Experiment 2 shows that there are quite a lot near duplicate comments — 564 from 602 comments in total included in JUnit4 project. Consequently, most comments in the considered project are near duplicates and just a few ones are unique. It shows that copy-pastes are normal in comment development for real-life industrial projects, although some additional experiments should be performed.

The run time of all algorithms on the JUint4 project data is shown in the Time column of Table 3. D2V and WMD have shown the worst time of all algorithms (50 and 30 minutes respectively). It should be noted that JUNit4 is not a large project. Therefore, this time is unacceptable for practical usage. D2V is slow since it computes its weights "on-the-fly", whereas all other algorithms use weights computed during the training phases. The variant of the WMD algorithm we used stalled Earth Mover's Distance metric computation, and in general its complexity is estimated as $O(p^3 \cdot log\ p)$, where p is the number of unique words in the compared text fragments. LCS, COS, and SNN run times range from 15 seconds to 1.5 minutes, which is acceptable for practical use.

LCS, COS, and WMD have shown the best results over the both experiments. However, WMD significantly loses in run time to string matching algorithms. Furthermore, LCS has shown the best performance, so we consider it to be the best choice for practical usage. This conclusion corresponds to the one made in [22], although the set of the algorithms used in their study differs from ours. LSH did not perform very well since Javadoc comments are smaller than DITA topics considered in [22]. This is the reason why hashed signatures turned out to be not so efficient.

6. CONCLUSION

We have analyzed similarity algorithms on the task of similarity of Javadoc comments. We have found out that existing algorithms are well-suited for this task. Our experimental results show that LCS (a string matching algorithm) is the best choice. It seems that combining supervised methods with domain-specific features and classic string matching algorithms could provide a more suitable result than pure machine learning. This question needs further research, as it seems that ML algorithms have a large potential, which, however, requires a lot of effort to be developed.

We have managed to find several copy-paste errors during data analysis: variations of some comment pairs that were not valid or, in other cases, they should have been but are absent. We can devise creating approaches and tools based on similarity algorithms for finding copy-paste errors as a direction for further research. Finally, it also important to improve the quality of comment similarity calculation algorithms by adding new domain-specific features and training them against larger amounts of input data that is better balanced regarding to the number of positive and negative comment pairs.

References

1. Spinellis, D. Code Documentation / D. Spinellis // IEEE Softw. — 2010. — Vol. 27, no. 4. — P. 18–19.

2. Oumaziz, M. A. Documentation Reuse: Hot or Not? An Empirical Study / M. A. Oumaziz et al. // Proc. of ICSR 2017. — 2017. — P. 12–27.

3. Blasi, A. Replicomment: identifying clones in code comments / A. Blasi, A. Gorla // Proc. of ICPC 2018, Gothenburg, Sweden. — ACM, 2018. — P. 320–323.

4. Nosál, M. Reusable software documentation with phrase annotations / M. Nosál, J. Porubän // Central Eur. J. Comput. Sci. — 2014. — Vol. 4, no. 4. — P. 242–258.

5. Corazza, A. On the Coherence between Comments and Implementations in Source Code / A. Corazza et al. // EUROMICRO-SEAA. — 2015. — P. 76–83.

6. Chin, F. Y. L. Binary Codes Capable of Correcting Deletions, Insertions and Reversals / F. Y. L. Chin, C. K. Poon // A fast algorithm for computing longest common subsequences of small alphabet size. — 1991. — Vol. 13(4). — P. 463–469.

7. Manning, C. D. Introduction to information retrieval / C. D. Manning et al. — 2008. — Vol. 1.

8. Gionis, A. Similarity Search in High Dimensions via Hashing / A. Gionis et al. // Proc. of VLDB 1999, Edinburgh, Scotland, UK. — Morgan Kaufmann, 1999. — P. 518–529.

9. Broder, A. Z. On the resemblance and containment of documents / A. Z. Broder // Compression and Complexity of Sequences 1997. Proceedings. — IEEE, 1997. — P. 21–29.

10. Kusner, M. J. From Word Embeddings To Document Distances / M. J. Kusner et al. // Proc. of ICML 2015, Lille, France. — Vol. 37. — JMLR.org, 2015. — P. 957–966.

11. Le, Q.V. Distributed Representations of Sentences and Documents / Q.V. Le, T. Mikolov // ICML 2014. — Vol. 32 of *JMLR Workshop and Conference Proceedings*. — 2014. — P. 1188–1196.

12. Mueller, J. Siamese Recurrent Architectures for Learning Sentence Similarity / J. Mueller, A. Thyagarajan // Proc. AAAI, 2016. — AAAI Press, 2016. — P. 2786–2792.

13. Tan, S. H. @tComment: Testing Javadoc Comments to Detect Comment-Code Inconsistencies / S. H. Tan et al. // ICST 2012. — IEEE Computer Society, 2012. — P. 260–269.

14. Fluri, B. Do Code and Comments Co-Evolve? On the Relation between Source Code and Comment Changes / B. Fluri, M. Würsch, H. C. Gall // Proc. of WCRE 2007, Vancouver, BC, Canada. — IEEE Computer Society, 2007. — P. 70–79.

15. Luciv, D. Detecting Near Duplicates in Software Documentation / D. Luciv, D. Koznov, G. Chernishev et al. // Programming and Computer Software. — 2018. — Vol. 44, no. 5. — P. 335–343.

16. Wen, F. A large-scale empirical study on code-comment inconsistencies / F. Wen et al. // Proc. of ICPC 2019 / Ed. by Y. Guéhéneuc et al. — IEEE / ACM, 2019. — P. 53–64.

17. Wang, D. Deep Code-Comment Understanding and Assessment / D. Wang et al. // IEEE Access. — 2019. — Vol. 7. — P. 174200–174209.

18. Zhou, Y. Analyzing APIs documentation and code to detect directive defects / Y. Zhou et al. // Proc of the ICSE 2017, Buenos Aires, Argentina. — IEEE / ACM, 2017. — P. 27–37.

19. Ratol, I. K. Detecting fragile comments / I. K. Ratol, M. P. Robillard // Proc. of ASE 2017, Urbana, IL, USA. — IEEE Computer Society, 2017. — P. 112–122.

20. Otaibi, J. A. Machine Learning and Conceptual Reasoning for Inconsistency Detection / J. A. Otaibi et al. // IEEE Access. — 2017. — Vol. 5. — P. 338–346.

21. Koznov, D. V. Clone Detection in Reuse of Software Technical Documentation /

D. V. Koznov et al. // 10th International Andrei Ershov Informatics Conference, PSI 2015. — Vol. 9609 of *LNCS*. — Springer, 2015. — P. 170–185.

22. Soto, A. J. Similarity-Based Support for Text Reuse in Technical Writing / A. J. Soto et al. // Proc. of the ACM DocEng 2015, Lausanne, Switzerland / Ed. by C. Vanoirbeek, P. Genevès. — ACM, 2015. — P. 97–106.

23. Luciv, D. Duplicate finder toolkit / D. Luciv, D. Koznov, G. Chernishev et al. // Proc. of ICSE 2018: Companion Proceeedings. — 2018. — P. 171–172.

24. Wagner, S. Analyzing Text in Software Projects / S. Wagner, D. M. Fernández // The Art and Science of Analyzing Software Data / Ed. by Christian Bird et al. — Morgan Kaufmann / Elsevier, 2015. — P. 39–72.

25. Basit, H. A. Efficient token based clone detection with flexible tokenization / H. A. Basit et al. // Proceedings of the ESEC/SIGSOFT FSE, 2007. — 2007. — P. 513–516.