Available online at www.ijpe-online.com

International Journal of Performability Engineering

vol. 15, no. 8, August 2019, pp. 2071-2080
DOI: 10.23940/ijpe.19.08.p6.20712080

Bug Report Classification based on Vector Space Model

Lele Chen, Song Huang*, Jinlei Sun, Zhanwei Hui, and Sen Yang

Command and Control Engineering College, Army Engineering University of PLA, Nanjing, 210007, China

Abstract

As a vehicle for recording and tracking defects, bug reports provide a basis for solving software quality problems. Currently, software testing is often carried out in a multi-person and parallel state. The integration process of numerous bug reports, such as moving fake or duplication bug reports, is facing severe challenges. Therefore, this paper proposes an automatic detection modus for bug reports based on the vector space model. After pre-processing the bug report, a matching library is created according to the test requirements and test report samples. The vector space model is used to calculate the similarity between the two, and the correctness of the bug report is detected based on this. Experiments with the data of a software test contest show that the modus proposed in this paper can correctly judge most bug reports, effectively improving the efficiency of de-false and de-duplication.

Keywords: software testing; vector space model; bug report; quality; natural language processing

(Submitted on June 10, 2019; Revised on July 12, 2019; Accepted on August 16, 2019)

1. Introduction

In the process of developing and maintaining large open-source software projects such as Eclipse [1] and Firefox [2], the bug tracking system is often used to record and manage defects. According to statistics, Firefox, Eclipse, and other large projects receive nearly 100 new bug reports each day [3]. Developers need to process the submitted bug reports and determine whether they contain new bugs. However, for these large software defect reporting libraries, nearly a third of the bug reports are marked by developers as duplicate bug reports [4].

The same problem also appears in the crowdsourced testing platforms. However, the manual evaluation of bug reports is very tedious and inefficient work. Therefore, how to automatically evaluate the quality of bug reports has become a topic worthy of study. To automatically determine whether the bug report submitted by a user is a duplicate bug report or a false bug report, based on the research by Thomas et al. [5], this paper proposes an automatic detection method for bug reports based on the vector space model. This method realizes the automatic evaluation of bug reports through modeling and classification. We have enhanced the efficiency of detecting bug reports. The organizational structure of this paper is as follows: Section 2 introduces the related background and technology. The method presented in this paper is explained in Section 3. Section 4 designs two experiments and analyzes the experimental results. In Section 5, we summarize and discuss the future work.

2. Related Background and Technology

2.1. Related Research

In recent years, research hotspots in bug reports have focused on the detection of duplicate bug reports. Related research is as follows:

Runeson et al. studied the detection of duplicate bug reports for the first time [6]. They used Sony Ericsson Mobile Communications Corporation's bug report database as their experimental data set. Firstly, the text information of the bug

* Corresponding author.
E-mail address: huangs_0317@126.com

reports was pretreated by vectorization and normalization, and then the similarity between the bug reports was calculated. The accuracy of the experimental results could reach about 30%. Based on the research of Runeson et al., Wang et al. [7] combined the software's execution information to give two definitions of bug report similarity: natural language similarity and execution information similarity. The experimental achieved great results in terms of recall and precision.

Jalbert et al. proposed a method to classify bug reports by using a linear regression classifier [8]. The input features of the training classifier were title similarity, description text similarity, clustering information, software version, and report submission date. The recall rate of this method was up to 50%. Kaushik and Tahvildari [9] compared the performance of the detection of repeated problem reports based on vector space models and topic-based models. The article selected three topic models: LSI [10-11], LDA [12], and random projections [13]. For the vector space model, cosine similarity was used to calculate the similarity, and various methods were tested to calculate the term weights. The experimental results showed that the method based on the vector space model is better.

Grineva et al. proposed a cluster-based method that represents a document in the form of a graph [14]. The edges and nodes of the graph were composed of the relationships between terms. The author considered that the more similar the topic of the document is, the more intensive the sub-graphs that are constructed. An algorithm for calculating the importance of subgraphs was proposed. Experiments showed that this method shows good results in a multi-topic and multi-noise document.

2.2. Information Retrieval Model

Information retrieval is a method for users to query and obtain relevant information from a collection of information according to their needs [15]. Generally, it is divided into broad information retrieval and narrow information retrieval. The general information retrieval models are the vector space model, latent semantic index, and latent Dirichlet allocation.

The vector space model (VSM) is a classical similarity calculation model. Documents are represented as vectors in document space by feature selection and weight calculation, and then the similarity between vectors is calculated to measure the similarity between documents. Cosine distance is the most common measure of vector space similarity in text processing.

The latent semantic index (LSI) is an extension of the VSM. Unlike the VSM, which uses precise word matching for text similarity calculation, the model considers the hidden meaning behind the word. For example, if the user searches for "method", the VSM only returns the page containing the word "method", but the user may also need a page containing the word "approach". The latent semantic index maps words and documents to a potential semantic space and compares the similarities of the documents in that space. The LSI derives the subject of the document based on the singular value decomposition (SVD) so that the problems of polysemy or multiple synonyms are solved.

The topic model is a statistical model for clustering implicit semantic structures of documents in an unsupervised learning manner [16]. It is mainly used in semantic analysis and text mining, and it is also applied in bioinformatics research [17]. The topic model is also a typical word bag model, as shown in Figure 1 [18]. A document is composed of many topics, and each topic is a collection of words. There is no ordering relationship between words. A typical topic model is latent Dirichlet allocation (LDA).

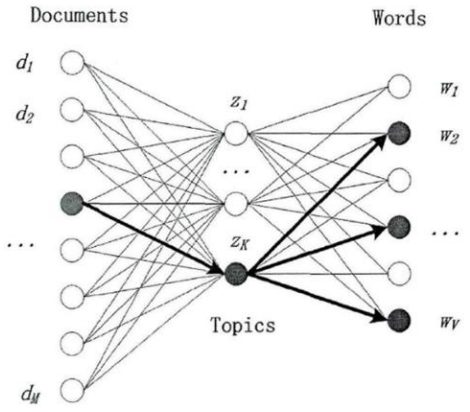

Figure 1. Document-topic-word three-layer model

2.3. Text Pre-Processing

Before the bug report is represented as a vector, some pre-processing of the text information is required, so that the characteristics of the text information can be clearly expressed. The general pre-processing work includes word segmentation, stemming, and removing stop words.

Word segmentation: Word segmentation is the decomposition of a sentence or a paragraph into a single word and punctuation. Take the sentence "I want to see a movie." as an example. After word segmentation, we will obtain "I", "want", "to", "see", "a", "movie", and "." Depending on the requirements, the results are expressed in the form of either an array or a document.

Stemming: The purpose of stemming is to normalize the different forms of the same word. In the document, some words often change due to singular, plural, or tenses. For example, the words "swam", "swum", "swims", and "swimming" will become the word "swim" after stemming. Commonly used methods for stemming based on grammar are the Porter stemmer [19] and Snowball stemmer [20].

Removing stop words: To save storage space and improve search efficiency, words that are automatically filtered when processing natural language data are called stop words. Stop words generally include two categories: non-realistic qualifiers, such as "the", "a", "an", "that", "those", etc., and those that are used widely but insufficient to express information about the document separately. In addition to some common stop words, developers typically build new stop word lists based on their data characteristics.

3. Bug Report Quality Detection Method based on the Vector Space Model

In this section, a quality detection method for bug reports based on the vector space model is proposed, and it is divided into four steps: firstly, pre-process the bug report. Secondly, model the bug report by using the vector space model. Thirdly, according to the test requirements, establish a bug report matching library. Finally, calculate the report similarity and compare it with manual review methods. Figure 2 shows this process.

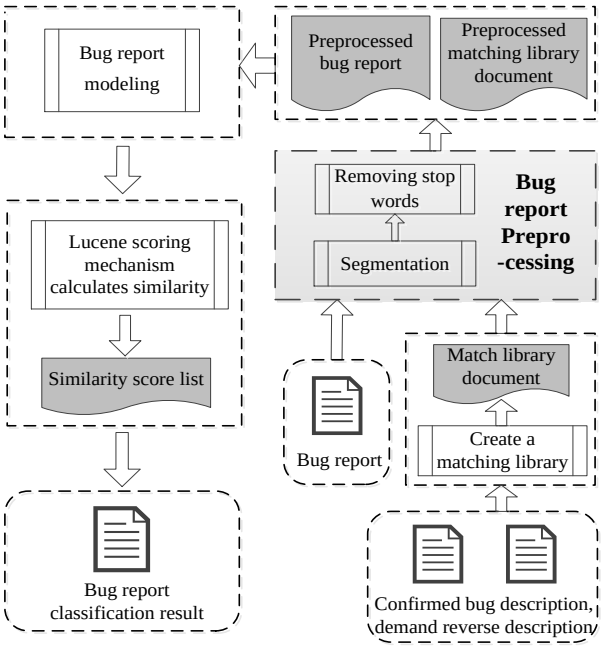

Figure 2. Bug reports' classification process

3.1. Bug Report Pre-Processing

The pre-processing of the bug report consists of two tasks: extracting the bug report text and pre-processing the text. Figure 3 shows the page for creating a bug report in Bugzilla. The information in a bug report usually contains the product, component, version, summary, description, and so on. When you submit the bug report, you will also receive a BugID.

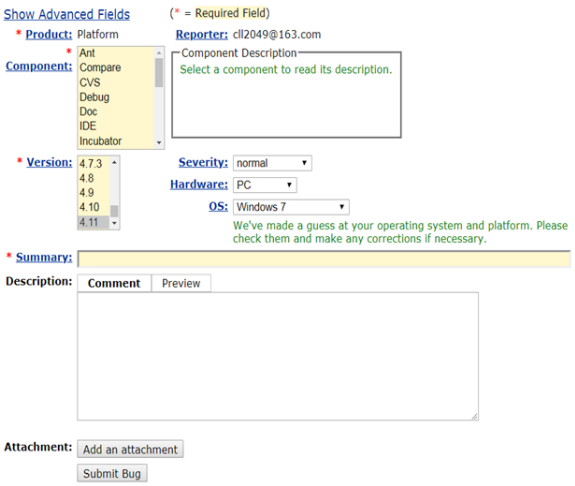

Figure 3. The page for creating a bug report in Bugzilla

Firstly, it is essential to extract the critical segment content, such as BugID and bug description, from the bug report, store it in the .txt document, and then perform pre-processing such as word segmentation, stemming, and removing stop words on the content of the document. Disabling the thesaurus usually requires adjustments based on the software's requirements specification. For example, for the bug "The left turn signal is on all the time", if the standard stop words library is used, "on" will be removed as a preposition. Then, the original meaning of this bug description will be changed, so in this case, stop words such as "on" and "not" cannot be deleted.

3.2. Bug Report Modeling

The vector space model was proposed by Salton et al. [21] in the 1970s and applied successfully in the SMART text retrieval system for the first time. This model simplifies the processing of text information into the operation of vectors in vector space. Each document can be represented as an n-dimensional vector. For example, document *D* can be expressed as $D = \{d_1, d_2, d_3, \cdots, d_n\}$, where d_i represents the weight of the i^{th} word in document *D*, that is, the importance of the term in the entire document. Different weight calculation methods will affect the text similarity. The standard weight calculation method is term frequency-inverse document frequency (TF-IDF).

The specific calculation method is as follows: given words w_i, the calculation formula of $tf_{i,j}$ in the document D_j is as follows:

$$tf_{i,j} = \frac{n_{i,j}}{\sum_k n_{k,j}} \tag{1}$$

Where $n_{i,j}$ represents the frequency of word w_i appearing in the document D_j and $\sum_k n_{k,j}$ represents the total number of words contained in D_j.

The formula for calculating the IDF is given as

$$idf_j = \log \frac{|D|}{|\{d : w_i \in d\}|} \tag{2}$$

Where $|D|$ represents the total number of documents contained in the collection and $|\{d : w_i \in d\}|$ represents the number of documents containing the word w_i in the group. Thus, the formula for calculating the $tf - idf_{i,j}$ of the word w_i is

$$tf - idf_{i,j} = tf_{ij} \cdot idf_i \tag{3}$$

After the document vectors are specified, the document similarity can be expressed by calculating the similarity

between vectors. The three conventional vector similarity calculation methods are cosine similarity, Dice similarity, and Jaccard similarity.

3.3. Create a Matching Library

This paper mainly studies how to remove false bug reports and duplicate bug reports on the newly submitted report under the premise of known software defects. Therefore, when establishing bug reports matching the library, it is necessary to include the confirmed bug description and the reverse description of the software requirements.

The primary purpose of adding the requirements of the reverse description to the bug report matching library is as follows: some false bug reports that are unrelated to the software can be removed while calculating the similarity, and it is helpful to discover new bugs in the software. When multiple bug reports are similar to a particular reverse requirement description, but the reverse requirement description is not confirmed as a software defect, developers will judge whether it is a new bug by reproducing the bug or viewing the code.

The basic approach is as follows: suppose a requirement described as A first obtains $\neg A$ by taking its negation. Because of the natural language characteristics represented in the bug report, take the same sentence B of $\neg A$. Domain experts complete this process, and finally $\neg A$ and B are added to the bug report matching library. When the speed V is more significant than zero, the data output interface "speed direction" is "right line". When the rate V is less than zero, the data output interface "speed direction" is "left line". When V is equal to zero, the data output interface "speed direction" is "stop". The reverse description of the demand for this paragraph is shown in Table 1.

Table 1. Reverse description of demand

No.	Description of requirement	Reverse description of requirements
1	When the speed *V* is more significant than zero, the data output interface "speed direction" is "right line".	When the speed *V* is more significant than zero, the data output interface "speed direction" is "left line".
2		When the speed *V* is greater than zero, the data output interface "speed direction" is "stop".
3	When the speed *V* is less than zero, the data output interface "speed direction" is "left line".	When the speed *V* is less than zero, the data output interface "speed direction" is "right line".
4		When the speed *V* is less than zero, the data output interface "speed direction" is "stop".
5	When the speed *V* is equal to zero, the data output interface "speed direction" is "stop".	When the speed *V* is equal to zero, the data output interface "speed direction" is "right line".
6		When the speed *V* is equal to zero, the data output interface "speed direction" is "left line".

3.4. Calculating Similarity

The text similarity calculation method used in this paper inherits the scoring mechanism in Lucene [22]. Lucene is an open-source full-text search engine toolkit, but it is only a full-text search engine architecture and not a complete full-text search engine. Lucene's purpose is to make it easier for developers to implement full-text search and build full-text search engines.

Lucene scores each document according to a scoring mechanism and returns a list of results sorted by descending order of score. Its scoring formula is given as

$$score(q,d) = coord(q,d) \cdot queryNorm(q) \cdot \sum_{t \in q} (tf(t \in d) \cdot idf(t)^2 \cdot t.getBoost() \cdot norm(t,d)) \tag{4}$$

Where q is the query, d is the document, t is the term, and each item in the formula represents a function that affects the score. The result of multiplying is the final score of the article. The meanings of each function in the formula are as follows:

$coord(q,d)$: Indicates the number of times the specified query item appears in the document. The number of times reflects the matching degree of the article. The higher the matching degree, the higher the score.

queryNorm(q) : This function is used to calculate the normalized value of each query and can compare different queries, but it does not affect the score ordering of the document. Its calculation formula is

$$queryNorm(q) = queryNorm\left(sumOfSquaredWeights\right) = \frac{1}{sumOfSquaredWeights^{\frac{1}{2}}} \tag{5}$$

$tf(t \in d)$ *and* $idf(t)$: These two sub-functions are the same as the weight calculation method introduced in the vector space model in Section 3.2, so they will not be repeated here.

t.getBoost() : This function sets the weight to Term during indexing and can statically and individually weight the specified domain or document. For example, when querying the term "Java", set the boost to 100, and then the weight matching "Java" will become higher.

norm(t,d) : This term is a weighting of the length, and the shorter the field (or the lower the vocabulary), the higher the weight gain. The value of this item is multiplied by *doc.getBoost* (the weight of the document), f*.getBoost* (the weight of the field), and *lengthNorm* (the number of *t* in *d*, which is generally used reciprocally). If the first two items are set to 1, then the value of *norm*(t,d) is equal to the value of *lengthNorm* . This paper defines some Java class based on Lucene's scoring mechanism and rewrites the above functions, making the score more reliable. This score is used as an essential reference for document similarity discrimination.

4. Experimental Analysis

In this section, we first inspect the effectiveness of the proposed method and then compare and analyze the traditional manual evaluation methods in terms of time and judgment efficiency.

4.1. Experimental Data

The data in this paper comes from all the problem reports submitted by contestants in the embedded item competition of a software testing competition in the recent three years. Each year's competitions are divided into summer qualifying competition, autumn qualifying competition, provincial competitions, and finals. Since the summer qualifying competition and the autumn qualifying competition use the same examination question, a total of nine examination questions and 7,422 bug reports were collected. The competition provides a document of design requirements and the corresponding software under test. The contestants design test cases and write test scripts according to the requirements document, find the embedded bug in the software by running test scripts, record the bugs, and submit bug reports. The traditional manual review method invites several software test experts to review the bug reports filed by the contestants. Table 2 shows the collected data.

Table 2. The experimental data

Year	Qualifying competition	Provincial competitions	Finals
2016	118/566	93/775	30/217
2017	168/2075	71/494	46/335
2018	178/1523	134/971	52/466

Firstly, it is necessary to extract the critical segment content, such as BugID and bug descriptions, from the bug report, store it in the .txt document, and then perform pre-processing such as word segmentation, stemming, and removing stop words in the content of the document. Disabling the thesaurus usually requires adjustments based on the software's requirements specification. For example, for the bug "The left turn signal is on all the time", if the shared stop words library is used, "on" will be removed as a preposition. Then, the original meaning of this bug description will be changed, so in this case, stop words such as "on" and "not" cannot be deleted.

4.2. Instance Verification

In this section, nine groups of data described in Table 2 are tested to verify the effectiveness of the proposed method. The steps of the experimental are as follows: firstly, the text pre-processing tool in document [5] is used to pre-process the bug

reports. Secondly, the requirements document of the test question is reversely described to form a document, and the matching database of the question report is established by integrating the standard answers. Then, the result of the pre-processing is modeled based on the vector space model, and the similarity of the bug reports is calculated using the method described in Section 3.4. Finally, the matches with the highest similarity score are selected to judge whether the matches are the standard answers based on the results of expert marking.

The evaluation of the bug report is a two-category problem. The real category of the problem and the type predicted by the method in this paper can be divided into the following four cases: true positive, false positive, true negative, and false negative. Let *TP*, *FP*, *TN*, and *FN* correspond to them respectively, and then *TP* + *FP* + *TN* + *FN* = total (the total number of bug reports) will be obtained. Table 3 shown the "confusion matrix" of the classification result.

Table 3. Confusion matrix

Real category	The category predicted by the method in this paper	
	Real bug	Fake bug
Real bug	*TP*	*FN*
Fake bug	*FP*	*TN*

This experiment selects the recall, precision, accuracy, false positive rate, and false negative rate as evaluation indicators. The specific definition and calculation methods are as follows:

Recall: Represented by *R*, this indicator is used to calculate the proportion of the number of real bugs queried by the method proposed in this paper to the actual number of real bugs in the bug report. The formula is

$$R = TP/(TP + FN) \times 100\% \tag{6}$$

Precision: Represented by *P*, this index is used to calculate the proportion of the number of real bugs queried by this method to the total number of queries. The formula is

$$P = TP/(TP + FP) \times 100\% \tag{7}$$

Accuracy: Represented by *A*, the indicator indicates the proportion of bugs with correct classification to the total number of bugs, which can reflect the ability of the classification method to determine the entire sample. The formula is

$$A = (TP + TN)/(TP + FP + TN + FN) \times 100\% \tag{8}$$

Missing rate: Represented by *E*, this indicator reflects the proportion of the number of real bugs that are missing in the quality report process and the actual number of real bugs in the bug report. The formula is

$$E = FN/(TP + FN) \times 100\% \tag{9}$$

False positive rate: Represented by *D*, this indicator reflects the proportion of the number of real bugs that are misjudged in the quality report process and the actual number of real bugs in bug report. The formula is

$$D = FP/(TP + FN) \times 100\% \tag{10}$$

4.3. Comparative Experiment

The system and application software involved in the performance comparison experiment is a 64-bit win7 system, Intel(R) Core(TM) i5-6200U CPU, Eclipse Java EE IDE for Web Developers with version Mars.1 Release, and JDK1.8.

The time used for automated evaluation includes the time of data pre-processing, the time to calculate the similarity, and the time taken to obtain the automatic method by writing a function that calculates the running time of the program. The traditional manual review method invites ten experts in the field of software testing to review the bug reports submitted by the contestants. The time it takes for an expert to review a bug report is about ten minutes, so we multiply the number of bug reports by ten to obtain the time for manual review.

In this experiment, the evaluation efficiency is used as the performance of the evaluation method. The evaluation efficiency, which is expressed by *T*, calculates the number of correct evaluation bugs within the unit time (min). The formula of the automatic method is shown as Equation (11), and the formula of the manual evaluation method is shown as Equation (12).

$$T\_A = TP / t \tag{11}$$

$$T\_M = (TP + FP) / t \tag{12}$$

Where *t* is the time used for the evaluation method. *TP* and *FP* are described in detail in Table 3 above, so they will not be described here.

4.4. Experimental Results and Analysis

Table 4 shows the experimental results of the validity of the quality detection methods for bug reports.

Table 4. Validity experiment result

Year	Evaluation index				
	R	*P*	*A*	*E*	*D*
	Qualifying competition				
2016	41.10%	63.40%	65.55%	58.90%	23.73%
2017	44.00%	79.28%	60.67%	56.00%	11.50%
2018	45.60%	85.95%	57.52%	54.40%	7.46%
	Provincial competitions				
2016	48.10%	83.21%	62.32%	58.90%	23.73%
2017	42.53%	86.55%	54.86%	57.47%	6.61%
2018	49.62%	86.47%	60.66%	50.38%	7.76%
	Finals				
2016	39.82%	72.58%	60.83%	60.18%	15.04%
2017	43.10%	86.55%	54.63%	56.90%	6.69%
2018	40.85%	52.25%	71.29%	59.15%	37.75%
Average	43.86%	77.36%	60.85%	56.14%	13.98%

As shown in Table 4, for the data of the three software test contests, the average values of the five evaluation indicators obtained by the method proposed in this paper are as follows: the recall rate is 43.86%, the precision is 77.36%, the accuracy is 60.85%, the missing rate is 56.14%, and the false positive rate is 13.98%. The results show that the proposed method has high precision. Through long-term observation of the three-year qualifier data, we can find that the recall and precision have an upward trend, while the missing rate and false positive rate show a downward trend.

The reason for the analysis is the following: with the test contest and related training year by year, the overall level of the contestants is gradually improved, and the corresponding bug discovery ability and the accuracy of the description of the bug are continuously improved. The more standardized the bug description, the higher the accuracy of the method for judging the bug report. This phenomenon was verified once again in the data of the 2018 provincial competition. The recall of 49.62% and the precision of 86.47% were the maximum values of the nine pieces of experimental data.

However, the final data of 2018 shows that both the recall and the precision indexes are not ideal. The analysis found that the final issue in 2018 is complicated, and the demand for the test piece is not easy to understand, resulting in a decline in the quality of the problem report submitted by the contestants. This affected the recall and precision.

The performance test results of time and evaluation rates are analyzed. Table 5 shows the results.

The experimental results show that the evaluation of the automated method takes about 0.31 seconds, which is much smaller than the manual review time. This value is the program run time divided by the number of bug reports per batch, taking the average of multiple times. Moreover, the automated method can correctly evaluate 48.50 reports per unit time, while the manual method can only judge 0.46 reports. Therefore, for a large number of bug reports, the automated review

method is significantly better than the manual review in terms of time and the evaluation efficiency.

Table 5. Performance experiment results

Method	Automated		Manual review	
Evaluation index	*t*(min)	T_A(QTY)	*t*(min)	T_M(QTY)
2016	2.84	34.12	1180	0.20
	4.10	55.55	930	0.51
	1.16	38.90	300	0.38
2017	10.70	49.72	1680	0.72
	2.84	52.06	710	0.49
	1.92	53.58	460	0.52
2018	7.64	62.47	1780	0.59
	4.96	65.71	1340	0.49
	2.37	24.44	520	0.27
Average	/	48.50	/	0.46

In this paper, the automatic detection method and manual evaluation method are analyzed on the real data set from five evaluation indexes: recall, precision, accuracy, missing rate, and false positive rate. The experimental results show that the recall and precision of the method are about 43% and 77% respectively, and it has a low false positive rate. At the same time, the advantages of the automation method are highlighted in terms of time and the evaluation rate, which also proves the validity and superiority of the quality report method based on the VSM model proposed in this paper.

5. Conclusions

In this paper, we proposed an automatic detection method for bug reports based on the vector space model. Firstly, the bug report was divided into word segmentation, stemming, and removing stop words. Secondly, the bug report matching library was built according to the test requirements and test report samples. Thirdly, the bug report was modeled, and the similarity was calculated using the vector space model. Finally, the correctness of the bug report was detected based on the similarity score. The feasibility and effectiveness of the method were proven by experiments in the data set of a software test contest.

In addition, the method proposed in this paper still has certain limitations. For example, when pre-processing the bug report, we do not know which words need to be removed to maximize the original meaning of the bug report and ensure the accuracy when calculating the similarity. In the future, it is necessary to conduct in-depth research on the pre-processing of bug reports to further improve the recall rate and precision of bug reports.

Acknowledgements

The project is supported by the National Key Research and Development Program of China (No. 2018YFB1403400).

References

1. Eclipse, "The Platform for Open Innovation and Collaboration," (https://www.eclipse.org/)
2. Firefox, "A Browser Built by Mozilla," (http://www.mozilla.org/en-US)
3. N. Battenburg, "Duplicate Bug Reports Considered Harmful. Really?" in *Proceedings of International Conference on Software Maintenance*, pp. 337-345, Beijing, China, October 2008
4. J. Anvik, L. Hiew, and G. C. Murphy, "Who Should Fix This Bug?" in *Proceedings of the 28th International Conference on Software*, pp. 361-370, Shanghai, China, May 2006
5. S. W. Thomas, "The Impact of Classifier Configuration and Classifier Combination on Bug Localization," *IEEE Transactions on Software Engineering*, Vol. 39, No. 10, pp. 1427-1443, October 2013
6. P. Runeson, M. Alexandersson, and O. Nyholm, "Detection of Duplicate Defect Reports using Natural Language Processing," in *Proceedings of International Conference on Software Engineering*, pp. 499-510, IEEE Computer Society, Minneapolis, MN, USA, May 2007
7. X. Wang, L. Zhang, and T. Xie, "An Approach to Detecting Duplicate Bug Reports using Natural Language and Execution Information," in *Proceedings of the 30th International Conference on Software Engineering*, pp. 461-470, Leipzig, Germany, May 2008
8. N. Jalbert and W. Weimer, "Automated Duplicate Detection for Bug Tracking Systems," in *Proceedings of the IEEE International Conference on Dependable Systems & Networks with Ftcs & Dcc*, pp. 52-61, Anchorage, AK, USA, September 2008

9. N. Kaushik and L. Tahvildari, "A Comparative Study of the Performance of IR Models on Duplicate Bug Detection," in *Proceedings of the Euromicro Conference on Software Maintenance and Reengineering*, 2012
10. S. Deerwester, S. T. Dumais, and G. W. Furnas, "Indexing by Latent Semantic Analysis," *Journal of the Association for Information Science & Technology*, Vol. 41, No. 6, pp. 391-407, May 2010
11. T. K. Landauer, D. S. Mcnamara, and S. Dennis, "Handbook of Latent Semantic Analysis," *Wiley Interdisciplinary Reviews Cognitive Science*, Vol. 4, No. 6, pp. 683-692, June 2014
12. D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent Dirichlet Allocation," *Journal of Machine Learning Research*, Vol. 3, pp. 993-1022, March 2003
13. P. Kanerva, J. Kristofferson, and A. Holst, "Random Indexing of Text Samples for Latent Semantic Analysis," in *Proceedings of the 22nd Annual Conference of the Cognitive Science Society*, pp. 103-106, Philadelphia, USA, May 2000
14. M. P. Grineva, M. N. Grinev, and D. Lizorkin, "Extracting Key Terms from Noisy and Multi-Theme Documents," in *Proceedings of the 18th International Conference on World Wide Web*, pp. 661-670, Madrid, Spain, April 2009
15. Information Retrieval, "The Method and Means of Finding Information," (http://en.wikipedia.org/ Information_ retrieval)
16. C. H. Papadimitriou, P. Raghavan, and H. Tamaki, "Latent Semantic Indexing: A Probabilistic Analysis," *Journal of Computer and System Sciences*, Vol. 61, No. 2, pp. 217-235, January 1998
17. B. Zheng, D. C. Mclean, and X. Lu, "Identifying Biological Concepts from a Protein-Related Corpus with a Probabilistic Topic Model," *BMC Bioinformatics*, Vol. 7, No. 58, pp. 147-156, December 2006
18. H. M. Wallach, "Topic Modeling: Beyond Bag-of-Words," in *Proceedings of International Conference on Machine Learning*, pp. 977-984, Pennsylvania, USA, June 2006
19. O. Baysal, M. W. Godfrey, and R. Cohen, "A Bug You Like: A Framework for Automated Assignment of Bugs," in *Proceedings of the IEEE 17th International Conference on Program Comprehension*, pp. 297-298, Vancouver, Canada, June 2009
20. D. Matter, A. Kuhn, and O. Nierstrasz, "Assigning Bug reports using a Vocabulary-based Expertise Model of Developers," in *Proceedings of the 6th International Working Conference*, pp. 131-140, Vancouver, Canada, May 2009
21. G. Salton, "A Vector Space Model for Automatic Indexing," *Communications of the ACM*, Vol. 18, No. 11, pp. 613-620, November 1975
22. M. Mccandless, E. Hatcher, and O. Gospondnetic, "Lucene in Action," 2nd Edition, 2010