Recognize steps to reproduce at multiple levels of granularity

Jia Zhang
University of Minnesota
zhan7164@umn.edu

Pranay Patil
University of Minnesota
patil122@umn.edu

ABSTRACT

Android developers get a lot of bugs reported on their apps in a very frequent manner. Going through these bug reports and identifying the reproduction steps to localize the bug is a time consuming task. The developer based on the contents of the report and his/her knowledge of app context, try to search the codespace for the correct methods to reproduce. We are automating this process with the help of NLP techniques such as VSM and semantic matching. We tested our approach on NextCloud, TeamAmaze, PocketHub, K-Mail applications to see the performance of our techniques.

1 INTRODUCTION

Android ecosystem has over 2 billion users and devices. It is not rare for the software systems to have bugs and android apps are not an exception to this. But with such a big ecosystem, number of bugs reported can be a lot.

For making bug reporting easy, every application project takes help of some bug reporting system such as Github issues, Bugzilla. These bugs can be reported in multiple formats. Most of them are in the form of bug reports. Here bug reports are a form of software lifecycle artifacts which is used for software maintenance. In general it consists of summary about the issue faced, steps to reproduce the issue, expected/observed behavior and sometimes stack trace. The table 1 shows the amount of bugs being reported and fixed for some of the most popular android applications[13].

After the bug is reported, based on developers' current assignments and the bug's priority it gets assigned to some developer. After which he/she analyzes the bug report's content. First thing is to reproduce and localize the cause of the issue which later he/she fixes and the fix is released. Now, the source code for these android applications can be huge, and manually localizing the reproduction steps is not trivial without a fair amount of existing context about the source's architecture and components.

Given such a high numbers of reported bugs and ever creasing complexity of android apps it is necessary to automate fault localization, to improve the efficiency of developers and let them focus more on how to fix rather than finding where to fix.

Some intro to studied papers

Unlike the existing work, we are trying to get more granular results. Our technique not only tries to search the file but also the method corresponding to a step required for reproduction. The main focus of our work is to utilize comments related to individual methods. Methods are attributed by comments, which describe the overall functionality of the method. We are using VSM and semantic matching for this purpose. In these technique we represent the comments and bug report in a vector space or a word embedding space to compute similarities between them. The technique is discussed in details in section 4.

2 BACKGROUND

2.1 VSM - vector space model

It is used to translate a text document in a vector space. In the Vector Space Model (VSM), each document or query is a N-dimensional vector where N is the number of distinct terms over all the documents and queries.The i-th index of a vector contains the score of the i-th term for that vector. The main score functions are based on: Term-Frequency (tf) and Inverse-Document-Frequency(idf).

The Term-Fequency (tf_{ij}) is computed with respect to the i-th term and j-th document:

$$tf_{ij} = \frac{n_{ij}}{\sum_k n_{kj}}$$

The Inverse-Document-Frequency (idf_i) takes into consideration the i-th terms and all the documents in the collection. The intuition is that rare terms are more important that common ones.

$$idf_i = \log \frac{|D|}{|d : t_i \epsilon d|}$$

The final score $w_{i,j}$ for the i-th term in the j-th document consists of a simple multiplication : $tf_{ij} * idf_i$. In order to compute the similarity between two vectors : a, b (document/query but also document/document), the cosine similarity is used:

$$\cos(a, b) = \frac{ab}{\|a\|\|b\|}$$

2.2 Word embedding

The common problem with lexical or pattern matching is that they fail to capture similarity between two semantically similar tokens. Word embedding fixes this issue, by translating tokens into such a vector space where similar tokens are closer to each other. In addition to this, numerous mathematical computations can be done on text document with the help of embedding. Word2Vec [4] and GloVe [6] are two popular and widely used methods to construct vector representations for words. Word2Vec is a two-layer neural network and an unsupervised algorithm for learning the meaning behind words. It produces a dense vector space that gives each word in the corpus a vector such that similar words are located close to each other in that space. GloVe is also an unsupervised algorithm for constructing word vectors. It uses word occurrences in a corpus as its source of information and builds a word-word co-occurrence matrix and then employs matrix factorization to build word vectors. And just as VSM, cosine similarity is used to compute similarity between two tokens.

3 MOTIVATING EXAMPLE

Here is a successful example of the method finder. This is issue 5995 of Github repository NextCloud[1], which is an Android file manager

[1] github.com/nextCloud/android

Project	Reported Bugs	Fixed Bugs	Time span	Fix Rate
Android	64,158	3,497	11/07-12/13	5.45%
Firefox	11,998	4,489	9/08-12/13	37.41%
K-9 Mail	6,079	1,200	6/10-12/13	19.74%
Chrome	3,787	1,601	10/08-12/13	42.28%
OsmAnd Maps	2,253	1,018	1/12-12/13	45.18%

Table 1: Projects examined, bugs reported/fixed, time span

app.
The issue title is "Crash when uploading", and it provides steps to reproduce. the steps are,
1. Set large folder to auto upload;
2. See the app crash continuously.
The semantic similarity result suggests two methods corresponding to each of the steps. One is **updateToAutoUpload** method, which would add files or folders to auto upload list; another is **DecryptedPushMessage** method, which is an unrelated method but was suggested due to some similarity in the name.
The VSM similarity result also suggests the **updateToAutoUpload** method, but it also suggests **setAutoUploadInit**, which is in the same file as the previous method. It doesn't provide useful information for the 2nd step, either.
We could see that the first instruction is clearly given. It mentioned large folder and auto upload. the "auto upload" is an important key words because it doesn't appear many times in the code, so if it is the key word, it would be much easier to locate. This is a great example of how this system is working. If we can locate methods for instructions like this, it would be helpful. The developer could deliberately write instructions containing key words he/she is looking for, and this system could intelligently match the same or similar keywords. This machinery is better than simple searching, and could enhance the debugging process.
But for the second step: This step is not related to any actions or methods. It simply tells the programmer that the app crashes, so it won't link to any methods. This is an example of invalid sentences.

4 TECHNIQUE

We are using two techniques, one is VSM similarity and semantic matching. In this section we will discuss these techniques and preprocessing of the source and bug reports.

4.1 Preprocessing

Comments extractions

In the pre-processing phase comments from the source repository are extracted. For this purpose, a java application which given the path to the source, parses all the java files under src/ path and extracts methods and their corresponding comments, was built using the library JavaParser [8]. The tool does not only extracts comments but also the variable used and method parameters. The resulted mapping is saved in a json keyed with the path to the function as follows:

```
{
    "path_to_file": {
        "package.class": {
            "method_name": {
                "variables": [],
                "comments": [],
                "methodName": "",
                "attributes": []
            }
        }
}
```

For example:

```
{
    "focus-android-master/app/src/focusRelease/java": {
        "org.mozilla.focus.utils.AdjustHelper": {
            "setupAdjustIfNeeded": {
                "variables": [
                    "config"
                ],
                "comments": [
                    "// RELEASE: Enable Adjust - This class has different
implementations for all build types.",
                    "// noinspection ConstantConditions"
                ],
                "methodName": "setupAdjustIfNeeded",
                "attributes": [
                    "application"
                ],
            }
        }
}
```

Reproduction step extraction from bug report

One of the most crucial and nontrivial part of this technique is to extract steps to reproduce the bugs. This doesn't have a trivial solution because these is no standard way to specify these steps. Bug reports are unstructured data types, identifying and extracting information of interest is tricky. For this purpose, we are using regular expression matching technique with following regex:

$$[0-9] + [.)\}] + [a - zA - Z'"0 - 9., ; : /?] +$$

This regex extracts text starting from numbers or letters which are most commonly used for indexing steps. The regex was tested on few bug reports and it worked for all of them.

Pre-processing on text

After extracting comments and steps from source files and bug reports, the pre-processing phase applies several steps to extract

desired word tokens. First we remove all the trailing white-spaces and punctuation marks. The words are then tokenized using nltk python library. Stop-words and java keywords are removed. In addition to this, camel cased words are separated into multiple tokens and added to the corpus.

4.2 VSM matching

A variation of VSM, rVSM[14] is used for computing similarity between the methods text and bug reports text. We use the approach similar to the one used in [2]. The weight w_{id} of term i in document d is calculated by the term frequency–inverse document frequency (tf–idf) score:

$$w_{id} = (1 + log(t_{id})) * (log(\frac{\#methods}{df_i}) + 1)$$

tf_{id} denotes the term frequency of the token i in a document d, #methods is the total number of methods in the corpus and df_i is the number of documents that term i appears in. Yhe resulting tf–idf vectors are then normalized by the Euclidean norm:

$$V_{norm} = \frac{V}{\sqrt{w_1^2 + w_2^2 + ... + w_v^2}}$$

Using this vectors similarity between bug reports and comments can be computed using cossine similarity:

$$cos(r, c) = \frac{V_r.V_c}{|V_r||V_c|}$$

V_r and V_c are the vector of term weights for bug report r and method comments c.

4.3 Semantic matching

It has been seen that the bug reports and the methods might not use the same tokens or common terms. This hinders the performance of a simple pattern matching VSM algorithm.

To deal with this issue we use semantic similarity to allow the reports and comments have some lexical gap. We use 300-dimensional pre-trained GloVe word vectors that are trained on the Common Crawl corpus[6]. This embedding is used to vectorize the tokens of the corpus. Since we are using pre-trained embedding, the tokens are kept unstemmed. Vectors of the report and comments are made by taking average of all the vectors of the document.

$$V_d = \frac{1}{N} \sum_{i=1}^{N} v_i$$

Using this vectors similarity between bug reports and comments can be computed using cossine similarity, as done in the VSM component:

$$cos(r, c) = \frac{V_r.V_c}{|V_r||V_c|}$$

V_r and V_c are the vector of term weights for bug report r and method comments c.

5 EVALUATION

We manually select 4 repositories' steps to reproduce, and check if they lead to valid results. This part is explained in *Evaluation method*. Our method could find the correct method names ideally, which is explained in *Evaluation results*. Qualitatively speaking, the result is not promising but reasonable; but there exists undesirable situations that cause problems, which we will talk about in *Revealed problems* section.

5.1 Evaluation method

We propose to evaluate our approach by manually checking what percentage of "steps" inside the issue could lead to valid method results. We use manual evaluation because there's no data set known useful for this particular evaluation. There exists a benchmark dataset provided by Zhou et al.(2012)[14], which is commonly used by many papers (e.g. Rahman et al.(2015)[7] and Gharibi et al.(2018)[2]). This benchmark contains 3 repositories namely the AspectJ, Zxing, and SWT, and the repositories' list of methods corresponding to fixed bug reports. This could be helpful when we are looking for the corresponding bug locations of the issues; however, what we are looking for is the method executed during the reproducing process, which may not be exactly the buggy code. The bug could happen elsewhere and lead to this problem, but the buggy code is not explicitly executed. There is no known dataset for our specific process (which is, uncovering corresponding methods executed for each issues with steps to reproduce). This could be a future direction of development, and this dataset could be of great help of this research area.

5.2 Evaluation results

The evaluation result is in the table 2. We tested on 4 repositories, with 27 issues and 101 steps to reproduce ("steps") in total. Among the 101 steps, 57 steps are valid. We explain what steps counts as valid in the next part *Revealed problems*. The retrieval is calculated by counting the valid results of VSM similarity, as VSM similarity result always contain the semantic valid result (and is higher than semantic result). The total retrieval percentage is about 60%, considering only the valid steps; and 34% if calculating all the steps. This tool is generally a helper which could help programmers locate executed files quicker, so the retrieval rate doesn't need to reach 100%, but the higher, the better. If we could eliminate the false positives in the list by giving every options confidence probability, it could save time for programmers from filtering out the misleading hints. We also evaluated whether tokens' information is useful. We compare "comments+tokens" and "comments only" and test whether comments only would give more promising result. Table 3 shows what percentage of methods have decent comments. There aren't much comments available, even if these are the repositories we hand-picked to ensure there're enough comments data to use. The result of the evaluation is, "comments only" work when the instruction is clear and simple, but it cannot find more complex or more unrelated information. For example, for the step of "Go to repository", both could locate "viewRepository" method. But for "Click to new an issue", only "comments+tokens" located the correct corresponding onClick method. The speed does not vary much between these two approaches, so taking the tokens into consideration is recommended.

Project	Issues	Sentences	Valid sentences	Semantic: Valid results	VSM: Valid results	Success retrieved %	Success retrieved %: valid sentences
NextCloud	11	36	25	17	19	52%	76%
K-9 Mail	6	22	15	6	7	32%	47%
TeamAmaze	6	29	7	4	5	17%	57%
PocketHub	4	14	10	3	3	21%	30%

Table 2: Evaluation result

Project	Total methods	Total Comments	Commented Methods	Commented Methods %
NextCloud	3,561	1,883	694	19.5%
K-9 Mail	3,732	957	327	8.76%
TeamAmaze	1,790	835	322	18%
PocketHub	646	33	16	2.48%

Table 3: Comments/Methods distribution

5.3 Revealed problems

Usually the issues' steps are not well-formed. The steps can be completely unrelated to the repository itself. The most common example would be steps like "Open an empty file". This step sentence locates in the AmazeFileManager, and actually means to create a new empty file ready to be edited by the user. The button is on the front page of the app, so this instruction is clear for users and programmers, but unclear for our approach to identify it because it contains too little information (too little key words). Our method rely heavily on more recognizable words. Words such as "night mode" are very effective in searching, but "Open", "empty", and "file" are existing everywhere inside the source code. Also, there are some other problems with the steps. For example, there exists steps like "Open the app", which is not exactly related to methods inside the app. There are steps like "Send a message in message app, and expects to see FileManager in the list". It is the bug of the repository being tested, but the problem happens outside the app, therefore the description is not helpful for finding the corresponding methods. There could also be some vague operations which doesn't contain any key words.

6 RELATED WORK

6.1 Papers

There exists many related work in this area which is also working on localizing relevant source files for steps to reproduce, issues, descriptions, etc. The methods vary, but most paper just use one or several methods, and linearly combine them together to optimize the final result. Ye et al.(2014)[11] proposes Learning to rank which could use API descriptions to link the bug report and the code. Sometimes the bug report and the code doesn't share the exact same word, but in API description they would be linked together (e.g. Bug report: Image. code: Picture. API: Image is picture.) Gharibi et al.(2018)[2] proposes a token matching component that tries to find exact matches of some specific textual tokens of bug reports in specific positions of source files. It also uses 4 other methods which are VSM, stack trace, semantic similarity, and a fixed bug reports component which uses previously fixed bug reports and a multi-label classification algorithm to score suspicious faulty source files according to existing solved bug reports. Lam et al.(2017)[1] provides a bug localization method which combines deep learning method into the conventional Information Retrieval methods. "DNNLOC" combines DNN, rVSM, and bug fix history, and they complement well to each other, and by suggesting 5 files could have a 70% coverage. Nguyen et al.(2011)[5] is almost the first work in this area. It provides bugscout which uses LDA to decide the possibility percentage of the relationship between the file and the topic. It trains the model with the historical data to estimate the topic distribution of a new bug report, and then compare it with all the source files, try to get the one with most similar topics. Kim et al.(2013)[3] provides a two-phase model which predicts if the bug report contains enough information for prediction. This method could be combined with our apporach to gain better and more reliable results. Zhang et al.(2017)[12] adds available metadata (such as component and version information) into the localization process to gain better results. Wang et al.(2020)[9] uses supervised topic modelling approach which makes observations and follow the observations to supervise the model. It uses a 5-step observation method. Wen et al.(2016)[10] provides Locus which could locate bugs from software changes. Changes in code has a finer granularity because it focuses on the changed part which could be exactly where the bug appeared. It provides important contextual clues for bug fixing.

6.2 Granularity

As for granularity, most approaches currently has a file granularity. Changing the granularity to method could be simple, but using file granularity would lead to better results as there is typically more information available per file. There is often very short methods, such like 5-6 lines, and we can hardly extract any information from a small method like this. Also, if there are too many methods inside a repository, it would be hard to find the exact method simply by checking the tokens; but when there are thousands of methods, there could be less than hundred of files which is easier to locate.

Technique names	Granularity	Summary
Learning to rank	Method	Use API descriptions to link the bug report and the code
5-Component bug localization	File	Linearly combine 5 techniques and giving each of them a weight
DNNLOC	File	Combine Deep Neural Network and IR methods
BugScout	File	Uses LDA which provides confidence level
Two-phase model	File level, "but easily could be extended"	Check if the bug report contains enough information for prediction; Proceed if yes
Exploring meta-data	File	After locating buggy files: Add available metadata to gain better results
Supervised model	File	Supervised topic modeling approach
Locus	Both the change and source file levels	Focus on where the code has changed, because it has finer granularity

Table 4: Related work

The granularity for each methods is present in the table 4. Fining the granularity is easy, but getting fine result is hard.

7 CONCLUSION & FUTURE WORK

In this project we tried to design an approach to automate the repro search. We have seen how it is crucial for developers to have this search more efficient. We proposed two approaches in which we are using semantic and VSM similarity over the method and bug report tokens. These tokens are built by parsing comments, parameters, variables from the method and repro steps from the bug report. In addition to the detailed implementation, we have also added quantitative comparison between the presented approaches. Furthermore, we also discussed qualitative impact of using additional tokens with the comments in terms of how it affects the precision.

Next, it would be interesting to see if we can integrate both approaches or we can use one approach to augment the other. One more interesting thing is to use word-embedding built upon Android code-bases for the semantic search. A possible but hard thought is, a dataset for this research area, which contains instruction steps and corresponding executed methods. It would be of great help in the researching area.

REFERENCES

[1] An Ngoc fLam, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N. Nguyen. 2017. Bug localization with combination of deep learning and information retrieval. In *Proceedings of the 25th International Conference on Program Comprehension, ICPC 2017, Buenos Aires, Argentina, May 22-23, 2017*, Giuseppe Scanniello, David Lo, and Alexander Serebrenik (Eds.). IEEE Computer Society, 218–229. https://doi.org/10.1109/ICPC.2017.24

[2] Reza Gharibi, Amir Rasekh, M. Sadreddini, and Seyed Fakhrahmad. 2018. Leveraging textual properties of bug reports to localize relevant source files. *Information Processing and Management* 54 (11 2018), 1058–1076. https://doi.org/10.1016/j.ipm.2018.07.004

[3] Dongsun Kim, Yida Tao, Sunghun Kim, and Andreas Zeller. 2013. Where Should We Fix This Bug? A Two-Phase Recommendation Model. *IEEE Trans. Software Eng.* 39, 11 (2013), 1597–1610. https://doi.org/10.1109/TSE.2013.24

[4] Tomas Mikolov, Ilya Sutskever, Kai Chen, G.s Corrado, and Jeffrey Dean. 2013. Distributed Representations of Words and Phrases and their Compositionality. *Advances in Neural Information Processing Systems* 26 (10 2013).

[5] Anh Tuan Nguyen, Tung Thanh Nguyen, Jafar M. Al-Kofahi, Hung Viet Nguyen, and Tien N. Nguyen. 2011. A topic-based approach for narrowing the search space of buggy files from a bug report. In *26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), Lawrence, KS, USA, November 6-10, 2011*, Perry Alexander, Corina S. Pasareanu, and John G. Hosking (Eds.). IEEE Computer Society, 263–272. https://doi.org/10.1109/ASE.2011.6100062

[6] Jeffrey Pennington, Richard Socher, and Christopher Manning. 2014. Glove: Global Vectors for Word Representation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics, Doha, Qatar, 1532–1543. https://doi.org/10.3115/v1/D14-1162

[7] S. Rahman, K. K. Ganguly, and K. Sakib. 2015. An improved bug localization using structured information retrieval and version history. In *2015 18th International Conference on Computer and Information Technology (ICCIT)*. 190–195.

[8] Danny van Bruggen, Federico Tomassetti, Malte Langkabel, Nicholas Smith, Artur Bosch, Malte Skoruppa, Cruz Maximilien, ThLeu, Panayiotis, Sebastian Kirsch (@skirsch79), Johann Beleites, Wim Tibackx, André Rouél, Daan Schipper, Why you want to know, Ryan Beckett, ptitjes, kotari4u, Ricardo Morais, Dominik Hardtke, bresai, Ty, Romain Lebouc, kgeilmann, dpnolte, Ayman Abdel Ghany, Enno Boland, Donny Nadolny, Björn Kautler, and Artem Konovalov. 2019. *javaparser/javaparser: v3.13.10.* https://doi.org/10.5281/zenodo.2667379

[9] Yaojing Wang, Yuan Yao, Hanghang Tong, Xuan Huo, Ming Li, Feng Xu, and Jian Lu. 2020. Enhancing supervised bug localization with metadata and stack-trace. *Knowledge and Information Systems* (1 Jan. 2020). https://doi.org/10.1007/s10115-019-01426-2

[10] Ming Wen, Rongxin Wu, and Shing-Chi Cheung. 2016. Locus: locating bugs from software changes. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, David Lo, Sven Apel, and Sarfraz Khurshid (Eds.). ACM, 262–273. https://doi.org/10.1145/2970276.2970359

[11] Xin Ye, Razvan C. Bunescu, and Chang Liu. 2014. Learning to rank relevant files for bug reports using domain knowledge. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, Shing-Chi Cheung, Alessandro Orso, and Margaret-Anne D. Storey (Eds.). ACM, 689–699. https://doi.org/10.1145/2635868.2635874

[12] Xiaofei Zhang, Yuan Yao, Yaojing Wang, Feng Xu, and Jian Lu. 2017. Exploring Metadata in Bug Reports for Bug Localization. In *24th Asia-Pacific Software Engineering Conference, APSEC 2017, Nanjing, China, December 4-8, 2017*, Jian Lv, He Jason Zhang, Mike Hinchey, and Xiao Liu (Eds.). IEEE Computer Society, 328–337. https://doi.org/10.1109/APSEC.2017.39

[13] Bo Zhou, Iulian Neamtiu, and Rajiv Gupta. 2015. A cross-platform analysis of bugs and bug-fixing in open source projects. *Proceedings of the 19th International Conference on Evaluation and Assessment in Software Engineering - EASE 15* (2015). https://doi.org/10.1145/2745802.2745808

[14] Jian Zhou, Hongyu Zhang, and David Lo. 2012. Where should the bugs be fixed? More accurate information retrieval-based bug localization based on bug reports. *Proceedings - International Conference on Software Engineering* (06 2012), 14–24. https://doi.org/10.1109/ICSE.2012.6227210