Reranking-based Crash Report Deduplication

★Akira Moroo ☆Akiko Aizawa ★Takayuki Hamamoto
★ Tokyo University of Science ☆ National Institute of Informatics

Abstract

Software projects collect and deduplicate vastly numerous crash reports from users to fix bugs efficiently. However, most existing automated methods have performance issues during large-scale clustering. We propose a reranking-based crash report clustering method. Our method is a combination of two earlier methods. By computing similarity used in ReBucket for the crash reports that are highly similar to the query crash report, the method can process reports with throughput equal to that of PartyCrasher. We also introduce an automatically generated dataset for crash report clustering tasks. The evaluation revealed that our method performs at high processing speed while maintaining high accuracy.

1 Introduction

Modern software products such as web browsers have a large and complicated code base along with an increasing number of bugs. To fix these bugs efficiently, developers prioritize high-frequency bugs using crash reports: summaries of the status of the software execution upon its unexpected termination. Figure 1 portrays the structure. A crash report includes a description, machine environment information, and a stack trace. A stack trace presents multiple stack frames with indexes. A lower index denotes a newer frame in the stack trace. An actual crash is caused in the #0 frame: the top frame. Therefore, a bug causing a crash is typically included in the frame near the top frame.

The information in stack traces is helpful for detecting and fixing bugs. Therefore, software-development projects collect such crash report from their users and cluster them according to the detected bugs. However, because of the numerous and diverse crash reports, clustering them manually is infeasible. Many automated crash report deduplication methods have been proposed to address this issue. However, most methods present performance issues because of their high computational costs. As described herein, we propose a reranking-based automated crash report clustering method. Reranking is a well-known technique for information retrieval where the ranking of initial search results is re-ordered using another computationally more expensive ranking function.

The contribution of this paper is that we introduce reranking-based crash report deduplication for reduced computational costs. This report is the first of the relevant

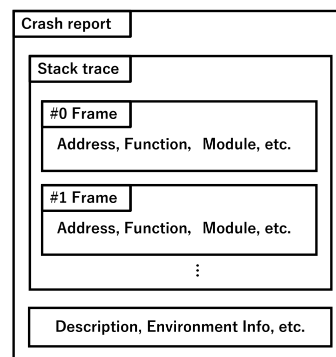

Figure 1: Crash Report Structure.

literature to describe a study by which a reranking scheme is shown to be effective in crash report deduplication.

2 Related Work

Existing methods for automatic crash report deduplication can be categorized into the following two types: One is based on the similarity of feedback from users written in natural language[1, 2]. The other is based on the similarity of stack traces. As described in this paper, we specifically examine the latter type because stack-trace-based methods achieve higher accuracy than NLP-based methods.

Socorro[1] is a crash report management system developed by Mozilla. In this system, a string called *signature* is generated for each crash report by applying several heuristic rules to the stack trace. Then, the system classifies all reports with the same signature into the same group. This system requires human elaboration for editing and updating signature-generation rules.

Lerch and Mezini [3] introduced a method using a TF-IDF-based full-text search engine. Campbell et al. [4] also used a full-text search engine and compared several tokenization methods used for indexing. These methods can achieve reasonable runtime performance for large-scale clustering. However, the accuracy is often worse than that achieved with other methods because the features used in the search engines ignore the order of the frames.

Dhaliwal et al. [5] proposed a method using the Levenshtein distance as similarity between two stack traces. The Levenshtein distance is used to measure the similarity of two strings. This approach regards one frame as a character. Dang et al. [6] defined similarity between two stack

[1]https://wiki.mozilla.org/Socorro

DOI reference number: 10.18293/SEKE2017-135

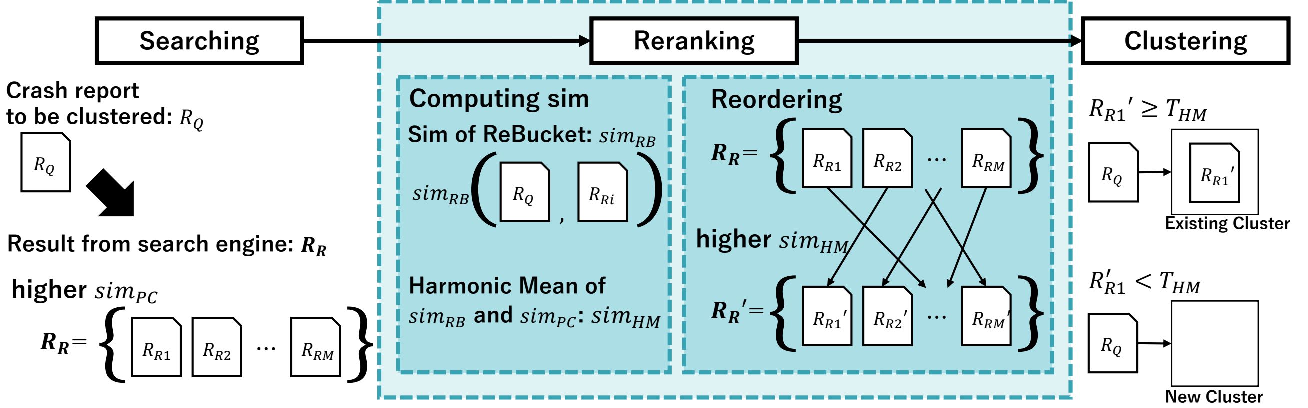

Figure 2: Overview of Proposed Method.

traces based on Position Dependent Model. Because these distance-based methods can capture the features of stack traces, they are useful for high-accuracy clustering but with high computational complexity.

As described in this paper, we use PartyCrasher [4] and ReBucket [6] as baselines. PartyCrasher is implemented as an open-source project[2]. It is easily extensible. Among these methods, ReBucket achieved the highest reported accuracy: 0.88 F1-measure on average.

3 Methodology

The proposed method is a consolidation of two state-of-the-art deduplication methods: ReBucket [6] and PartyCrasher [4]. Figure 2 presents an overview of the method. It comprises three modules: searching, reranking, and clustering. Given a pool of crash reports (already clustered) and a target crash report to be clustered, the search module uses the target report as a search query. Thereby, it obtains an initial ranking list of possibly similar crash reports. We follow the technique proposed in PartyCrasher [4] here, and use a general-purpose search engine. Next, the reranking module re-orders the initial results with the similarity function defined in ReBucket [6]. Finally, the clustering module either selects a cluster from existing ones or generates a new cluster for the target crash report.

3.1 Searching

Let R_Q be a crash report to be processed. Using a full-text search engine, the system obtains the top M search result $\boldsymbol{R}_R = \{R_{R1}, R_{R2}, \ldots, R_{RM}\}$. In [4], multiple tokenization methods for indexing are compared. Our system uses `Camel`, which achieved the highest accuracy. In `Camel`, all frames in a stack trace are tokenized by space separation and for each camel case. Symbols are deleted. Hereinafter, we designate the stack trace of the crash report R_X as C_X.

Most full-text search engines use TF-IDF weight to rank results. Term Frequency (TF) represents how many times a certain term appears in a document. Inverse Document Frequency (IDF) is the reciprocal of the number of occurrences of a certain word in all documents. In our method, we use the same normalization as the one used in Elasticsearch[3]. Letting $W(C_q)$ be a set of words in a stack trace C_q, tf_{t,C_d} and idf_{t,C_d} be a value of DF and IDF of a word t in a stack trace C_d, respectively, and denoting the word count in C_d as nt_{C_d}, the similarity score used in PartyCrasher between C_q and C_d is defined as

$$sim_{PC}(C_q, C_d) = \sum_{t \in W(C_q)} \left(\sqrt{tf_{t,C_d}} \times idf_t \times \frac{1}{\sqrt{nt_{C_d}}} \right). \tag{1}$$

3.2 Reranking

In ReBucket [6], similarity is computed by emphasis on the common frame position between two stack traces. Two metrics are defined: distance to the top frame and alignment offset. The distance to the top frame represents the distance from an arbitrary frame to the top frame in the same stack trace. The alignment offset represents the difference of the distance to the top frame between the frames that match in two stack traces.

These metrics are based on the following insights: First, an important frame has a smaller distance to the top frame. Second, the alignment offset between matching frames is smaller for stack traces with higher similarity. Let f_i and f_j respectively denote the i-th and the j-th frames of C_q and C_d. Denoting distance to the top frame as $min(i, j)$ and alignment offset as $abs(i - j)$, the cost function $cost(i, j)$

[2] https://github.com/naturalness/partycrasher

[3] https://www.elastic.co/products/elasticsearch

of f_i and f_j is defined as follows in ReBucket.

$$cost(i,j) = \begin{cases} e^{-c*min(i,j)} e^{-o*abs(i-j)} & if \;\; f_i = f'_j \\ 0 & otherwise \end{cases} \quad (2)$$

Therein, c and o are weighting parameters. With ReBucket, similarity between stack traces C_q and C_d is defined as

$$sim_{RB}(C_q, C_d) = \frac{M_{m,n}}{\sum_{j=0}^{l} e^{-cj}}, \quad (3)$$

where m and n are at respective depths of C_q and C_d, and $l = max(m,n)$. In addition, $M_{i,j}$ is the optimal cost at depth i, j calculated recursively using dynamic programming.

The computational cost is $O(mn)$ in the above calculation. According to [7], about 90 % of crash reports can be fixed by referring only to the top 10 frames. Therefore, we use only the top 10 frames in our calculation.

The final similarity score between C_q and C_d we use for reranking is given as

$$sim_{HM} = \frac{sim_{RB} \cdot sim_{PC}}{\alpha \cdot sim_{RB} + (1-\alpha) \cdot sim_{PC}}. \quad (4)$$

3.3 Clustering

In the same way as PartyCrasher, we compare the similarity of crash report R'_{R1}, which has the highest similarity in $\boldsymbol{R}'_R$, with the threshold T_{HM}. If it is greater than T_{HM}, then R_Q is clustered into the same cluster as R_{R1}. Otherwise, it is clustered into a new cluster because it is regarded as a crash by the new type bug.

The computational overhead of the search engine in our method is $O(NlogN)$, which is negligible compared with the computational cost of stack trace similarity. Therefore, we compare the size of the distance matrix that requires stack trace similarity calculation. ReBucket uses hierarchical agglomerative clustering (HAC) algorithm for crash report clustering. This method requires $O(N^2)$ because it computes an $N \times N$ distance matrix where N is the number of total crash reports. In contrast, the clustering algorithm used in the proposed method requires $O(MN)$, where M is the number of search results. As with $M \ll N$, the proposed method can reduce the computational cost compared with HAC.

4 Evaluation Experiment

In this section, we present evaluations of PartyCrasher, ReBucket, and our proposed method from the perspectives of accuracy and runtime performance.

4.1 Dataset

We use two datasets: `Launchpad`[4], an existing dataset from Ubuntu Launchpad, and `Firefox48`, a new dataset we collected from Mozilla Firefox. Table 1 presents a summary of the datasets.

[4]https://archive.org/details/bugkets-2016-01-30

Table 1: Dataset Information.

	Project	Crash reports	Clusters	Version	Packages
`Launchpad`	Ubuntu Launchpad	15,293	3,824	N/A	816
`Firefox48`	Mozilla Firefox	36,752	727	from 48.0a1 to 48.0b99	1

`Launchpad` is more reliable than `Firefox48` because `Launchpad` contains clusters produced by hand, whereas the clusters of `Firefox48` are automatically constructed as shown below. `Firefox48` differs from `Launchpad` in that the dataset includes a single software package whereas `Launchpad` includes many different packages provided by Ubuntu's package manager. Many projects deal with a single product in their report systems. Therefore, we presume that our evaluation can be more solid by introducing the `Firefox48` dataset.

We followed [5] to create the dataset, which comprises crash reports and corresponding bugs. A set of crash reports that link to the same bug form a cluster. We gathered 100 crash reports at most from each of the top 300 high-frequency clusters between ver. 48.0a1 and 48.0b99. When the same bug is found to cause two crash clusters, these clusters are grouped into a single cluster. Furthermore, we extract stack traces and metadata from the crash reports.

4.2 Evaluation Metrics

Let BCubed precision be denoted as Bp and BCubed recall as Br. Bp represents how many reports in an estimated cluster belong to the same cluster in ground truth. Br denotes how many reports in a cluster of grand truth belong to the same estimated cluster. The quality of the generated clusters is better when the values of Bp and Br are closer to 1.0. Based on the Bp and Br, the BCubed F1-meature (BCF) is calculated as $BCF = (2 \cdot Bp \cdot Br)/(Bp + Br)$. A tradeoff exists between Bp and Br. Higher BCF indicates higher accuracy[8].

The number of crash reports processed per unit of time is taken as the performance criterion. ReBucket uses multiprocessing to calculate distance matrixes. Therefore, we multiply the actual processing time by the number of processes.

4.3 Determining Parameters

We determined optimal parameters using a subset from the first 20 % of the dataset. The rest are used for testing. The time cost of determining optimal parameters is not included in the performance results. Table 2 presents the final values used in the evaluation. Both ReBucket and our method require c and o in (2). T_{RB} is a distance threshold for HAC in ReBucket. Moreover, α in (3), M, the size of initial ranking list, and T_{HM}, the clustering parameter, are

Table 2: Parameters used in the evaluation.

	c	o	T_{RB}	α	M	T_{HM}
Launchpad	0.3	0.1	0.07	0.3	50	11.0
Firefox48	0.4	0.4	0.26	0.0	50	2.0

Table 3: Evaluation Results of Accuracy.

	Launchpad			Firefox48		
	Bp	Br	BCF	Bp	Br	BCF
PartyCrasher	0.723	0.671	0.696	0.778	0.424	0.549
ReBucket	0.775	0.515	0.618	0.662	0.717	0.688
Proposed	0.807	0.626	0.705	0.728	0.651	0.687

Table 4: Performance Evaluation Results.

	Performance $[Reports/s]$	
	Launchpad	Firefox48
PartyCrasher	2.500	4.663
ReBucket	0.004	0.021
Proposed	2.922	4.022

used only for the proposed method. The number of used processes of ReBucket is 60 because of limitation of the available computation resources. The parameters c, o, and T_{RB} are robust and about 20 % of the searched combinations achieve over 90 % of the highest BCF. By contrast, α, M and T_{HM} are sensitive. Especially, we observe that when M is too large, both accuracy and performance decrease.

4.4 Results

Table 3 presents accuracy evaluation results. The proposed method achieved the highest BCF value for Launchpad. Bp of our method is also the highest. In addition, results show that ReBucket showed high accuracy in terms of BCF because the Br of PartyCrasher drops to 0.424. Moreover, the BCF value of the proposed method is the same as that of ReBucket.

The difference in accuracy between Launchpad and Firefox48 can be attributed to the differences of the numbers of packages: 816 for Launchpad and 1 for Launchpad. It is likely that the same cluster includes reports from the same package in most cases. Results show that ReBucket has a difficulty recognizing the difference when the dataset includes multiple packages because it relies completely on frames in the stack traces.

To validate this point, we created a subset of Launchpad using reports from only one package, "nautilus." Then, the BCF values of PartyCrasher, ReBucket, and our method become 0.616, 0.662, and 0.666, correspondingly. With the subset, ReBucket achieves better performance than PartyCrasher, which is the opposite of the result with the original Launchpad. Based on this result, we conclude that ReBucket does not perform well if multiple packages are included in the dataset. The proposed reranking scheme compensates the issue by filtering out unrelated candidates before applying ReBucket.

Table 4 presents performance evaluation results. Table 4 shows that the processing speed of the proposed method is higher than that of ReBucket. In addition, our method achieved almost equal performance to that of PartyCrasher. Our method is almost 730 times faster.

To summarize, our results demonstrate that the proposed method has better runtime performance than ReBucket while maintaining accuracy.

5 Conclusion

As described in this paper, we propose an automated crash report deduplication method applying reranking against reports of the results obtained from the full-text search engine. We also constructed a new dataset for crash report clustering tasks from Mozilla Firefox. The evaluation results demonstrate that our method is equal to that of ReBucket in terms of accuracy, but with faster performance.

Our current method uses statically tuned parameters with a subset. Manual merging or division of crash clusters is assumed in real operations. In future studies, we expect to investigate a dynamic parameter tuning method using user feedback for more stable accuracy.

References

[1] P. Runeson, M. Alexandersson, and O. Nyholm, "Detection of duplicate defect reports using natural language processing," in *Proceedings of the 29th International Conference on Software Engineering*, 2007, pp. 499–510.

[2] A. Alipour, A. Hindle, and E. Stroulia, "A contextual approach towards more accurate duplicate bug report detection," in *Proceedings of the 10th Working Conference on Mining Software Repositories*, 2013, pp. 183–192.

[3] J. Lerch and M. Mezini, "Finding duplicates of your yet unwritten bug report," in *Proceedings of the 2013 17th European Conference on Software Maintenance and Reengineering*, 2013, pp. 69–78.

[4] J. C. Campbell, E. A. Santos, and A. Hindle, "The Unreasonable Effectiveness of Traditional Information Retrieval in Crash Report Deduplication," in *Proceedings of the 13th International Conference on Mining Software Repositories*, 2016, pp. 269–280.

[5] T. Dhaliwal, F. Khomh, and Y. Zou, "Classifying Field Crash Reports for Fixing Bugs: A Case Study of Mozilla Firefox," in *Proceedings of the 2011 27th IEEE International Conference on Software Maintenance*, 2011, pp. 333–342.

[6] Y. Dang, R. Wu, H. Zhang, D. Zhang, and P. Nobel, "ReBucket: A Method for Clustering Duplicate Crash Reports Based on Call Stack Similarity," in *Proceedings of the 34th International Conference on Software Engineering*, 2012, pp. 1084–1093.

[7] A. Schröter, N. Bettenburg, and R. Premraj, "Do stack traces help developers fix bugs?" in *2010 7th IEEE Working Conference on Mining Software Repositories*, 2010, pp. 118–121.

[8] E. Amigó, J. Gonzalo, J. Artiles, and F. Verdejo, "A Comparison of Extrinsic Clustering Evaluation Metrics Based on Formal Constraints," *Inf. Retr.*, vol. 12, no. 4, pp. 461–486, 2009.