TraceSim: A Method for Calculating Stack Trace Similarity

Roman Vasiliev
JetBrains
St.Petersburg, Russia
roman.vasiliev@jetbrains.com

Dmitrij Koznov
Saint-Petersburg State University
St.Petersburg, Russia
d.koznov@spbu.ru

George Chernishev
Saint-Petersburg State University
St.Petersburg, Russia
g.chernyshev@spbu.ru

Aleksandr Khvorov
JetBrains, ITMO University
St.Petersburg, Russia
aleksandr.khvorov@jetbrains.com

Dmitry Luciv
Saint-Petersburg State University
St.Petersburg, Russia
d.lutsiv@spbu.ru

Nikita Povarov
JetBrains
St.Petersburg, Russia
nikita.povarov@jetbrains.com

ABSTRACT

Many contemporary software products have subsystems for automatic crash reporting. However, it is well-known that the same bug can produce slightly different reports. To manage this problem, reports are usually grouped, often manually by developers. Manual triaging, however, becomes infeasible for products that have large userbases, which is the reason for many different approaches to automating this task. Moreover, it is important to improve quality of triaging due to a large volume of reports that needs to be processed properly. Therefore, even a relatively small improvement could play a significant role in the overall accuracy of report bucketing. The majority of existing studies use some kind of a stack trace similarity metric, either based on information retrieval techniques or string matching methods. However, it should be stressed that the quality of triaging is still insufficient.

In this paper, we describe TraceSim — a novel approach to this problem which combines TF-IDF, Levenshtein distance, and machine learning to construct a similarity metric. Our metric has been implemented inside an industrial-grade report triaging system. The evaluation on a manually labeled dataset shows significantly better results compared to baseline approaches.

CCS CONCEPTS

• **Software and its engineering** → **Maintaining software**; *Software testing and debugging*; • **Information systems** → **Deduplication**.

KEYWORDS

Crash Reports, Duplicate Bug Report, Duplicate Crash Report, Crash Report Deduplication, Information Retrieval, Software Engineering, Automatic Crash Reporting, Deduplication, Crash Stack, Stack Trace, Automatic Problem Reporting Tools, Software Repositories.

MaLTeSQuE '20, November 13, 2020, Virtual, USA

ACM ISBN 978-1-4503-8124-6/20/11…$15.00
https://doi.org/10.1145/3416505.3423561

ACM Reference Format:
Roman Vasiliev, Dmitrij Koznov, George Chernishev, Aleksandr Khvorov, Dmitry Luciv, and Nikita Povarov. 2020. TraceSim: A Method for Calculating Stack Trace Similarity. In *Proceedings of the 4th ACM SIGSOFT International Workshop on Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE '20), November 13, 2020, Virtual, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3416505.3423561

1 INTRODUCTION

Systems for collecting and processing bug feedback are nearly ubiquitous in software development companies. However, writing bug reports may require substantial effort from users. Therefore, in order to reduce this effort, a way to create such reports automatically is implemented in most widely used products. In most cases, information available at the time of the crash, i.e. stack trace, is used to form a report.

The drawback of this approach is the huge number of generated reports, the majority of which are duplicates. For example, the study [12] describes WER — the system used in Microsoft to manage crash reports. This system collected billions of reports from 1999 to 2009. Another example is the Mozilla Firefox browser: according to the study [5], in 2016 Firefox was receiving 2.2 million crash reports a day.

It was demonstrated [9] that correct automatic assignment has a positive impact on the bug fixing process. Bugs whose reports were correctly assigned to a single bucket are fixed quicker, and, on the other hand, bugs with reports that were "spread" over several buckets take a longer time to fix.

Thus, the problem of automatic handling of duplicate crash reports is relevant for both academia and industry. There is already a large body of work in this research area, and providing its summary can not be easy, since different studies employ different problem formulations. However, the two most popular tasks concerning automatically created bug reports are:

(1) for a given report, find similar reports in a database and rank them by the likelihood of belonging to the same bug (ranked report retrieval) [4, 19];
(2) distribute a given set of reports into buckets (report clusterization) [11].

For both of these tasks, defining a good similarity measure is a must, since the quality of the output largely depends on it. Moreover, it is important to improve similarity algorithms carefully due to the big volume of reports that needs to be processed properly. Even

a relatively small improvement could play a significant role in increasing the quality of report bucketing.

In this paper, we address the problem of computing the similarity of two stack traces. The majority of deduplication studies can be classified into two groups: based either on TF-IDF or stack trace structure. The former use an information retrieval approach, while the latter employ string matching algorithms (such as edit distance) to compute stack trace similarity. However, to the best of our knowledge, there are no studies that offer a proper, non-straightforward combination of these two approaches. Such combination may result in a superior quality of bucketing and may significantly outperform any method that belongs to these individual groups. To substantiate importance of this idea, we would like to quote Campbell et al. [5]: "a technique based on TF-IDF that also incorporates information about the order of frames on the stack would likely outperform many of the presented methods...".

At the same time, machine learning (ML) was rarely applied to this domain: the majority of existing similarity calculation methods does not rely on ML techniques. The reason behind this is the fact that classic (non-ML) methods are more robust and stable than ML ones, which is very important for the considered task. Therefore, our idea is to use classic approaches as the basis.

However, ML methods are more flexible and their application has allowed to achieve substantial results in many areas of software engineering. Here, in this particular problem, moderately employing ML allows us to efficiently integrate both classic approaches. We believe that combining all three approaches would allow us to design a superior similarity function.

The contribution of this paper is TraceSim — the first algorithm for computing stack trace similarity that structurally combines TF-IDF [25] and string distance while using machine learning to improve quality.

We validate our algorithm using a real-life database of crash reports collected for JetBrains products.

2 BACKGROUND AND RELATED WORK

Systems that automatically collect crash stack traces from remote instances are widely used in mass-deployed applications. Prominent examples of such systems are Mozilla Socorro, LibreOffice Crash Reports, Google Chromium crash reporting system, Windows Error Reporting [7, 12] and many others. These systems act as additions to traditional bug trackers. They are tightly integrated with them in order to link stack traces to existing bugs, to form new bugs out of a collection of stack traces, and so on.

This kind of system allows to obtain bug feedback without requiring users to form and submit "classic" bug reports. This allows to increase the amount of collected feedback. Overall, the benefits of deploying such a system are the following:

- It allows to survey bug landscape at large at any given moment. For example, LibreOffice Crash Reports show[1] the aggregated view of all received reports over the last N days.
- It helps to locate the bug in the source code. Both Mozilla Socorro and LibreOffice Crash Reports are integrated with project repositories. The user can click on stack frames that are attached to a bug and be transferred to the corresponding lines in the source code.
- It allows to automate the assignment of bugs to developers. For example, ClusterFuzz[2] automatically assigns a bug to a developer based on the crash location in the source code.

All the use-cases mentioned above require to manage harvested stack traces which includes collecting, storing, and retrieving. For all these operations to be efficient, it is necessary to have the ability to compare stack traces with respect to the bugs that spawn them.

The challenge is not only the large number of reports, but also a widespread presence of exact and inexact duplicates. For example, our internal study has found that 72% of crash reports of the IntelliJ Platform (a JetBrains product) are duplicates. Due to a large volume of data, it is necessary to have a high-quality stack trace similarity measure. This, in turn, facilitates accurate elimination of duplicates and grouping of similar crash reports. Therefore, the choice of measure has a significant impact on ensuring the quality of the software product.

In our survey, we restrict ourselves to reviewing studies that present systems using an explicit similarity function for bucketing reports based on their stack traces. Surveys on triaging involving textual descriptions and tags can be found in [13, 22, 26].

Since we are interested in constructing a novel stack trace similarity measure that will combine TF-IDF, edit distance, and supervised machine learning approaches, we highlight the respective components of existing studies. The big picture is presented in Table 1.

Brodie et al. [4] were some of the first researchers who have addressed the problem of crash report deduplication using stack trace comparison. They present a biological sequence search algorithm that is a modification of the Needleman-Wunsch algorithm [21].

Bartz et al. [1] construct a callstack similarity measure that is essentially a modification of the edit distance metric, and propose seven edit operations with different weights.

Dhaliwal et al. [9] propose a two-step approach that combines signatures and the Levenshtein distance between stack traces. The idea is the following: first, reports are grouped together using only the first frame. Then, each bucket is split into several subgroups using the Levenshtein distance between stack traces (only the top 10 frames are used).

Kim et al. [14] use stack traces contained in a bucket, building a special graph on the base of similarity of two stack traces, and then applying a graph similarity measure to decide whether the new stack trace belongs to this bucket.

Modani et al. [19] compare three methods for calculating stack trace similarity: edit distance, prefix match and Longest Common Subsequence (LCS).

Dang et al. [7] present a new similarity metric that is based on the offset distance between the matched functions and the distance from these functions to the top frame. This method employs edit distance and relies on supervised learning approach.

Lerch and Mezini [15] study the situation when a bug tracker does not contain a dedicated field for storing crash stacks. The authors employ the TF-IDF approach for finding duplicate stack traces.

[1] https://crashreport.libreoffice.org/stats/

[2] https://google.github.io/clusterfuzz/

Table 1: Existing approaches

Method	TF-IDF	Edit Dist.	M. Learn.
Brodie et al. [4]		✓	
Bartz et al. [1]		✓	✓
Dhaliwal et al. [9]		✓	
Kim et al. [14]			✓
Modani et al. [19]		✓	
Dang et al. [7]		✓	✓
Lerch and Mezini [15]	✓		
Wu et al. [28]	✓		
Campbell et al. [5]	✓		
Moroo et al. [20]	✓	✓	✓
Sabor et al. [23]			✓

Wu et al. [28] adapt the TF-IDF approach by introducing the notions of function frequency and inverse bucket frequency. A notable idea of this approach is to expand the list of functions present in the stack trace by adding the ones that are likely to be the root cause of the crash. For this, a technique comprised of control flow analysis, backward slicing, and function change information is proposed.

Campbell et al. [5] compare automatic crash report deduplication methods. The authors have considered two types of algorithms: TF-IDF-based (using ElasticSearch [3]) and signature-based. The results of the evaluation demonstrate the superiority of information retrieval methods.

Moroo et al. [20] propose a reranking-based crash report clustering method. It is a combination of two state-of-the-art report deduplication methods: ReBucket [7] and Party-Crasher [5]. Since this method employs ReBucket as its part, it has both edit distance and supervised learning components. The Party-Crasher part supplies TF-IDF. However, authors propose a straightforward technique which essentially invokes these two approaches independently and then computes their weighted harmonic mean. While experiments demonstrated that such technique can be superior to its constituent parts, it is still not a proper structural integration. It is possible that an algorithm with "true" structural integration of TF-IDF and edit distance components (i.e. that describes more sophisticated relation between them) may yield significantly better results.

Finally, Sabor et al. [23] propose the DURFEX system which combines stack trace similarity and the similarity of two non-textual fields. For computing stack trace similarity, the authors propose to substitute function names by names of packages where they are defined and then to segment the resulting stack traces into N-grams of variable length.

We can see that despite this problem being studied for at least 15 years, it is still relevant for the community: new studies continue to emerge. Although there is a variety of methods, the quality of bucketing continues to be insufficient. Finally, there are no approaches that structurally integrate edit distance and TF-IDF, even though this combination looks promising [5] in a sense that it may substantially improve the quality of bucketing.

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

3 ALGORITHM

In this section, we describe our algorithm for computing stack trace similarity. Our algorithm takes two stack traces as its input. First, it processes stack overflow exceptions (SOEs) separately, since these stack traces contain a large number of repeated frames, and their similarity can be calculated effectively using TF-IDF (here we used approach from [15]). If the input stack traces are not SOEs, the algorithm proceeds to compute their similarity in a different way. First, it computes the weight for each frame of the stack traces, because different frames have different impacts on stack trace similarity. Next, the edit distance between two stack traces is calculated. In our approach, this distance is defined as Levenshtein distance [16] with frame weights. Finally, the results are normalized using the calculated Levenshtein distance. An implementation of our algorithm can be found here: https://github.com/traceSimSubmission/trace-sim.

A detailed description of the above steps follows.

3.1 Separate Processing of SOEs

A stack trace that is a stack overflow exception contains many repeated frames which refer to recursive calls. If this recursive part of two stack traces is similar, it is highly probable that they address the same error situation. Usually, this recursive part is rather large, significantly exceeding the non-recursive part of the stack trace in size. Therefore, complicated tests are unnecessary for such stack traces, and computing their closeness in terms of frame frequencies is enough. This is the reason we use the TF-IDF algorithm from [15] in this case.

3.2 Frame Weight Computation

While comparing two stack traces, differences in frames that are close to the top of the stack are usually more important than differences in deeper-positioned frames. This happens because upper frames more frequently contain the source of the bug (see RQ 2 in Schroter et al. [24]). We propose to represent this influence as frame weight: frames with higher weights are considered more important. We identify two factors that affect frame weight: frame position within a stack trace and frame frequency among all frames of all stack traces available in our database. For a stack frame f_i of $ST = f_0, \ldots, f_{N-1}$, its weight is calculated as follows:

$$\mathbf{w}(f_i) = \mathbf{lw}_\alpha(f_i) * \mathbf{gw}_{\beta\gamma}(f_i), \tag{1}$$

where $\mathbf{lw}_\alpha(f_i)$ is the local weight of f_i, i.e. the degree of its importance among other frames of the same stack trace, and $\mathbf{gw}_{\beta\gamma}(f_i)$ is the global weight of the frame, i.e. the degree of its importance among all frames of all stack traces presenting in our database. Here, α, β and γ are numeric hyperparameters [6] that are used to adjust the model to fit the data (i.e. to tune the algorithm to a particular stack trace collection).

Local frame weight of f_i is calculated as follows:

$$\mathbf{lw}_\alpha(f_i) = \frac{1}{i^\alpha} \tag{2}$$

Local weight is higher for frames which are closer to the top of the stack, since as practice shows, these frames are more important than further ones, i.e., errors are more likely caused by the functions which were called last.

Global frame weight of f_i is calculated according to the well-known information retrieval TF-IDF approach [18] as $\mathrm{TF}(f_i) * \mathrm{IDF}(f_i)$, where $\mathrm{TF}(f)$ (term frequency) represents the importance of the frame within a particular stack trace, while $\mathrm{IDF}(f)$ (inverse document frequency) represents how uncommon is the frame f for the whole corpus of stack traces. In our work, we do not use the TF part and consider it equal to 1 since it does not consider frame ordering, which is actually the most important information about the frame within the stack trace. This has already been taken into account when calculating $\mathbf{lw}_\alpha(f_i)$. Hence, we only calculate $\mathrm{IDF}(f_i)$ as

$$\mathrm{IDF}(f_i) = \log \frac{\text{Total num. of stack traces}}{\text{Num. of stack traces } ST : f_i \in ST}.$$

Therefore, we calculate global weight as follows:

$$\mathbf{gw}_{\beta\gamma}(f_i) = \sigma(\beta(\mathrm{IDF}(f_i) - \gamma)), \tag{3}$$

where σ is a sigmoid function defined as:

$$\sigma(x) = \frac{1}{1 + e^{-x}}. \tag{4}$$

Here, the β and γ hyperparameters are used to tune smooth filtering for $\mathrm{IDF}(f_i)$. We give small weights for very common frames that are contained in a large number of stack traces. Those can be frames that emerge due to frequently invoked chunks of code: commonly used development frameworks, logging or thread pooling.

3.3 Levenshtein Distance Calculation

In order to express difference between stack traces numerically, we use modified Levenshtein distance. As the basis we took classic Levenshtein distance that contains only insertion, deletion, and replacement operators. We do not consider a variation that includes transposition operation, since that for stack traces the order of the frames is very important: swapping places of two frames within a single stack trace is meaningless.

For two strings, classic Levenshtein distance is defined as minimal editing cost, i.e. the minimal total number of insertions, deletions, and replacements of a single character needed to transform one string into another [16]. For two stack traces, we define the distance in the same way, but additionally using the weights assigned to frames: stack traces that differ in "heavy" frames are more different themselves.

When calculating the cost of insertion, deletion or substitution of a frame, we define operation costs as follows: cost of insertion and deletion is the weight of the corresponding frame and the weight of substitution is the sum of weights of the original and the new frame.

3.4 Normalization

We do not use the Levenshtein distance itself for classification and clustering, instead, we calculate a normalized similarity value:

$$\mathrm{sim}(ST', ST'') = 1 - \frac{\mathrm{dist}(ST', ST'')}{\sum_{i=0}^{N'-1} \mathrm{w}(f_i') + \sum_{i=0}^{N''-1} \mathrm{w}(f_i'')}, \tag{5}$$

where $\mathrm{dist}(ST', ST'')$ stands for the Levenshtein distance between $ST' = f_0', \ldots, f_{N'-1}'$ and $ST'' = f_0'', \ldots, f_{N''-1}''$.

This step is needed to make interpretation and understanding of the similarity results easier. We believe that it is more comfortable to compare values belonging to $[0, 1]$ rather than to $[0, +\infty)$.

3.5 Hyperparameter Estimation via Machine Learning

In previous subsections, we have introduced α, β and γ — numeric hyperparameters used in calculation of local and global frame weights. To obtain their values we formulate an optimization problem and approach it with machine learning. The idea is to optimize the ROC AUC [10] metric by training on a manually labelled part of the stack trace dataset. These parameters are selected once for a dataset and are the same for each stack trace comparison. We have used the Tree-structured Parzen Estimator Approach (TPE) [2] and the hyperopt[4] [3, 6] library to solve this problem.

4 EVALUATION

4.1 Experimental Setup

To perform the evaluation, we have used the JetBrains crash report processing system Exception Analyzer which handles reports from various IntelliJ Platform products. Exception Analyzer receives generated reports and automatically distributes them into existing issues (buckets) or creates new ones out of them. However, it is a well-known problem that output of automatic bug triaging tools can be of insufficient quality [7]. Therefore, Exception Analyzer allows to employ user input to triage "problematic" reports. If this happens, user actions are logged and can be used later on for various purposes.

In order to evaluate our approach, we had to construct a test corpus. We adhere to the following idea: if a developer assigns a report to an issue manually, then there is a reason to think that this report is a duplicate to the ones already contained in the issue. And vice versa: if a developer extracts some reports from a given issue, it means that these reports are distinct from the remaining.

To construct our test corpus, we have extracted and analyzed reports from recent user action logs of Exception Analyzer spanning one year time frame. To create positive pairs we have analyzed user sessions and searched for the following pattern: for a particular unbucketed report, a user looks into some issue, compares it to a particular report of this issue and then assigns it into the issue. To obtain negative pairs we exploit a similar idea: we designate a pair as negative if a user compared reports and did not grouped them. Eventually, we have obtained 6431 pairs, out of which 3087 were positive and were 3344 negative. We have got not too many pairs due to the fact that most reports in Exception Analyzer are grouped automatically and users rarely have to intervene. The experiments were run with 80/20 train-test split.

4.2 Research Questions

RQ1: How do individual steps contribute to the overall quality of algorithm output? RQ1 evaluates the effectiveness of individual components of our method. Since our function consists of

[4]Hyperopt framework. https://github.com/hyperopt/hyperopt

a number of independent steps, it is necessary to check whether each of them is beneficial or not. By performing these evaluations, we demonstrate that each component is essential for our resulting similarity function. We perform several experiments for this purpose. For every experiment, we switch off the corresponding component in the full TraceSim, and run it on the test corpus. We consider the following steps: TraceSim without gw ($\mathbf{gw}(f_i) = 1$ in (1)), TraceSim without lw ($\mathbf{lw}_\alpha(f_i) = 1$ in (1)), TraceSim without SOEs (without separate processing of stack overflow exceptions using the algorithm from [15]), and the full version of TraceSim.

RQ2: How well does our approach perform in comparison with state-of-the-art approaches? RQ2 compares the resulting similarity function with the state-of-the-art approaches. First, we considered approaches that use TF-IDF [15, 20] technique. Next, we also employed Rebucket [7] method which belongs to edit distance and supervised learning methods. We have also included other edit distance methods in our baseline — Levenshtein distance [19] and Brodie [4]. Another supervised method that we have included in our evaluation is Moroo's et al. [20]. It combines Rebucket and Campbell's et al. [5] approaches. The latter was also included into our evaluation. All methods were implemented from scratch, except Rebucket [7] for which an existing implementation[5] was used.

We did not compare with recently-developed DURFEX [23] approach since it relies on tight integration with a bug tracker and requires component and severity fields. Our approach concerns only stack traces.

Finally, we have decided to compare our approach with several classic and widely known approaches: Prefix Match and Cosine Similarity [19]. We have employed two variations of the latter: Cosine Similarity with an IDF component (denoted as Cosine (IDF)) and without (denoted as Cosine (1)).

4.3 Evaluation Metrics

To answer RQs 1 and 2, we have evaluated how good our similarity function is. Due to the nature of our dataset, we have to use metric applicable for binary classification. To assess the quality of our algorithm, we use the well-accepted comparison measure ROC AUC [17]. It is statistically consistent, and it is also a more discriminating measure than Precision/Recall, F-measure, and Accuracy. Several studies concerning bug report triage also employ metrics like MAP [13], Recall Rate [8, 27], and other metrics used for the ranking problem. However, in this paper we consider the binary classification task and therefore we need to use other metrics.

It is important to note that 0.5 is considered the minimum result for ROC AUC due to simple random classifier giving a result of 0.5. In our experiments we did not use cross validation since we have a sufficient volume of data to run a simple test/train split.

Another observation is the following: if an algorithm increases ROC AUC from 0.5 to 0.55, this increase is less significant than the one from 0.75 to 0.8, despite the equal gain. This is the reason why we have computed the error reduction of each algorithm (RQ2). After ranging the algorithm outputs by ROC AUC, we calculate by how many percent the error rate has been reduced in comparison to the previous algorithm. For example, the method of Brodie et al. has improved by 0.06 in comparison to Prefix Match (0.64 against 0.58), and its error reduction is $0.06 * 100/(1 - 0.58) = 14\%$. These numbers are presented in Table 3.

[5]Implementation of Rebucket. https://github.com/ZhangShurong/rebucket

Table 2: Contribution of individual steps

Method	Results
TraceSim	0.79
TraceSim without SOEs	0.78
TraceSim without lw	0.76
TraceSim without gw	0.69

4.4 Results

RQ1: How do individual steps contribute to the overall quality of the algorithm output? The ROC AUC results are presented in Table 2. We have found out that the **gw** weight function, which is based on computing global frequency for frames, makes the largest contribution (+0.1). The **lw** weight function that considers the order of frames in a stack trace contributes less (+0.03). Finally, SOEs contribute the least (+0.01), which is explicitly connected to the number of stack traces that contain recursion (4% in our test corpus).

RQ2: How well does our approach perform in comparison to state-of-the-art approaches? The ROC AUC results are presented in Table 3. Our method turned out to be superior to all others. Our contribution is significant: we have improved by +0.03 compared to the existing algorithm with the best result on our dataset. However, it should be noted that the improvement of almost all other algorithms lies between +0.003 and +0.06. Furthermore, our algorithm provides error reduction of 13%, and only Brodie et al. provide more.

The following observations can be made based on these two tables. First of all, Table 3 shows that it is essential to use IDF for computing stack trace similarity. Methods that use IDF (Cosine, Lerch and Mezini, Campbell et al.) are all on the top of the list. Applying IDF is important for the task at hand since it takes into account which methods (functions) appear frequently in the stack trace corpus and which do not.

Next, the approach by Moroo et al. [20] has indeed turned out to be equal or superior to its all predecessors. However, on our dataset it has failed to surpass Campbell et al. [15], one of the two methods that it is built on.

We believe that this happened due to the inherent drawbacks of the approach. The algorithm of Moroo et al. straightforwardly combines the IDF technique and information on the stack trace structure: the authors invoke these two approaches independently and then compute the final score as their weighted harmonic mean.

On the other hand, TraceSim tries to provide structural integration of these two components. These components do not interact with each other in the Moroo's et al. algorithm, but they do in our approach: we compute weights for edit distance using IDF.

Another surprising result is the low score of Brodie's el al. method. It is a modification of Levenshtein distance which loses to the vanilla one on our dataset. The reasons become clear considering the specifics of the approach: this algorithm does not use machine learning and does not adjust to data. This demonstrates the need to employ machine learning if high performance is required.

Table 3: Comparison with other approaches

Similarity	ROC AUC	Error red.
TraceSim	0.79	13%
Moroo et al. [20]	0.76	0%
Campbell et al. [5]	0.76	0%
Lerch and Mezini [15]	0.76	11%
Cosine (IDF)	0.73	10%
Rebucket [7]	0.70	6%
Cosine (1)	0.68	0%
Levenshtein [16]	0.68	11%
Brodie et al. [4]	0.64	14%
Prefix Match [19]	0.58	–

Finally, consider Table 2 once more. Here, we can see that turning off gw leads to performance dropping to the level of Rebucket. This can be explained by the following: gw is essentially an IDF component, which is absent in Rebucket.

5 THREATS TO VALIDITY

We have identified the following threats to validity for our study:

- **Subject selection bias.** In our study, we use JetBrains product data only. Open source projects or projects of other companies may have different initial data. Experiments run on other data may yield different results. However, some of our results are corroborated by studies that examine the same subject [7, 9].
- **Quality of labeled data.** Since data labeling involves users it is prone to errors and may result in data of insufficient quality. The labeling procedure is described in Section 4.1. The concern is the fact that some of the reports in a bucket do not actually belong there. This may happen due to a lot of reasons: buckets are long-living objects and can accumulate "wrong" reports over time. To address this, we create positive pairs only from reports which a user had manually inspected before assigning a report. Additionally, we assume quality conscious labelling since it is a part of the product development workflow.
- **Significant stream of reports.** Our approach is intended for software products that have a large userbase and thus generate a large number of reports daily. If this is not true, our approach may not be optimal or needed at all. In some cases, manual report triaging could be a better choice.

6 CONCLUSION

In this paper, we have proposed a novel approach to calculating stack trace similarity that combines TF-IDF and Levenshtein distance. The former is used to "demote" frequently encountered frames via an IDF analogue for stack frames, while the latter allows to account for differences not only in individual frames, but also in their depth. At the same time, employed machine learning allowed us to efficiently combine two classic approaches.

To evaluate our approach, we have implemented it inside an industrial-grade report triaging system used by JetBrains. The approach has been employed for over 6 months, receiving positive feedback from developers and managers, who reported that the quality of bucketing had improved. Our experiments have shown that our method outperforms the existing approaches. It should be noted that even a relatively small improvement plays a significant role in the quality of report bucketing due to the large overall report volume.

REFERENCES

[1] K. Bartz et al. 2008. Finding Similar Failures Using Callstack Similarity *(SysML'08)*. USENIX Association, 1–6. http://dl.acm.org/citation.cfm?id=1855895.1855896

[2] J. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl. 2011. Algorithms for Hyper-Parameter Optimization *(NIPS'11)*. 2546–2554.

[3] J. Bergstra, D. Yamins, and D.D. Cox. 2013. Making a Science of Model Search: Hyperparameter Optimization in Hundreds of Dimensions for Vision Architectures *(ICML'13)*. JMLR.org, I–115–I–123.

[4] M. Brodie et al. 2005. Quickly Finding Known Software Problems via Automated Symptom Matching *(ICAC'05)*. 101–110. https://doi.org/10.1109/ICAC.2005.49

[5] J. C. Campbell, E. A. Santos, and A. Hindle. 2016. The Unreasonable Effectiveness of Traditional Information Retrieval in Crash Report Deduplication *(MSR '16)*. ACM, 269–280. https://doi.org/10.1145/2901739.2901766

[6] M. Claesen and B. De Moor. 2015. Hyperparameter search in machine learning. *arXiv preprint arXiv:1502.02127* (2015).

[7] Y. Dang, R. Wu, H. Zhang, D. Zhang, and P. Nobel. 2012. ReBucket: A Method for Clustering Duplicate Crash Reports Based on Call Stack Similarity *(ICSE '12)*. IEEE Press, 1084–1093. http://dl.acm.org/citation.cfm?id=2337223.2337364

[8] J. Deshmukh, K. M. Annervaz, S. Podder, S. Sengupta, and N. Dubash. 2017. Towards Accurate Duplicate Bug Retrieval Using Deep Learning Techniques *(ICSME '17)*. 115–124. https://doi.org/10.1109/ICSME.2017.69

[9] T. Dhaliwal, F. Khomh, and Y. Zou. 2011. Classifying Field Crash Reports for Fixing Bugs: A Case Study of Mozilla Firefox *(ICSM '11)*. IEEE Computer Society, 333–342. https://doi.org/10.1109/ICSM.2011.6080800

[10] T. Fawcett. 2006. An introduction to ROC analysis. *Pattern Recognition Letters* 27, 8 (jun 2006), 861–874. https://doi.org/10.1016/j.patrec.2005.10.010

[11] M. A. Ghafoor and J. H. Siddiqui. 2016. Cross Platform Bug Correlation Using Stack Traces *(FIT '16)*. 199–204.

[12] K. Glerum et al. 2009. Debugging in the (Very) Large: Ten Years of Implementation and Experience *(SOSP '09)*. ACM, 103–116. https://doi.org/10.1145/1629575.1629586

[13] A. Hindle and C. Onuczko. 2018. Preventing duplicate bug reports by continuously querying bug reports. *Empirical Software Engineering* (20 Aug 2018). https://doi.org/10.1007/s10664-018-9643-4

[14] S. Kim, T. Zimmermann, and N. Nagappan. 2011. Crash graphs: An aggregated view of multiple crashes to improve crash triage *(DSN '11)*. 486–493.

[15] J. Lerch and M. Mezini. 2013. Finding Duplicates of Your Yet Unwritten Bug Report *(CSMR '13)*. IEEE Comp. Soc., 69–78. https://doi.org/10.1109/CSMR.2013.17

[16] V. I. Levenshtein. 1966. Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Soviet Physics Doklady* 10 (1966), 707–710.

[17] C. X. Ling, J. Huang, and H. Zhang. [n.d.]. AUC: A Statistically Consistent and More Discriminating Measure Than Accuracy *(IJCAI'03)*. 519–524. http://dl.acm.org/citation.cfm?id=1630659.1630736

[18] C. D. Manning, P. Raghavan, and H. Schütze. 2008. *Introduction to Information Retrieval.* Cambridge University Press.

[19] N. Modani et al. 2007. Automatically Identifying Known Software Problems *(ICDEW '07)*. IEEE Computer Society, 433–441. https://doi.org/10.1109/ICDEW.2007.4401026

[20] A. Moroo et al. 2017. Reranking-based Crash Report Deduplication. In *SEKE '17*, X. He (Ed.). 507–510. https://doi.org/10.18293/SEKE2017-135

[21] S.B. Needleman and C.D. Wunsch. 1970. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology* 48, 3 (1970), 443–453. https://doi.org/10.1016/0022-2836(70)90057-4

[22] M. S. Rakha et al. 2018. Revisiting the Performance Evaluation of Automated Approaches for the Retrieval of Duplicate Issue Reports. *IEEE Trans. on Soft. Eng.* 44, 12 (Dec 2018), 1245–1268. https://doi.org/10.1109/TSE.2017.2755005

[23] K. K. Sabor et al. 2017. DURFEX: A Feature Extraction Technique for Efficient Detection of Duplicate Bug Reports *(ICSQRS '17)*. 240–250.

[24] A. Schroter et al. 2010. Do stack traces help developers fix bugs? *(MSR '10)*. 118–121. https://doi.org/10.1109/MSR.2010.5463280

[25] K. Sparck Jones. 1972. A statistical interpretation of term specificity and its application in retrieval. *Journal of documentation* 28, 1 (1972), 11–21.

[26] C. Sun, D. Lo, S. Khoo, and J. Jiang. 2011. Towards more accurate retrieval of duplicate bug reports *(ASE '11)*. 253–262.

[27] C. Sun, D. Lo, X. Wang, J. Jiang, and S. Khoo. 2010. A discriminative model approach for accurate duplicate bug report retrieval *(ICSE '10)*. 45–54.

[28] R. Wu et al. 2014. CrashLocator: Locating Crashing Faults Based on Crash Stacks *(ISSTA '14)*. 204–214. https://doi.org/10.1145/2610384.2610386